



XAPP197 (v1.0.1) July 6, 2006

Triple Module Redundancy Design Techniques for Virtex FPGAs

Author: Carl Carmichael

Summary

Triple Module Redundancy (TMR) combined with Single Event Upset (SEU) correction through partial reconfiguration is a powerful and effective SEU mitigation strategy. This method is only supported for the Virtex™ series of Xilinx FPGAs. Xilinx Application Note, XAPP216, describes the use of Readback and Partial Configuration for SEU detection and correction. This application note outlines the recommended design methodology for constructing and implementing TMR logic within the Virtex architecture.

TMR in FPGAs

Introduction

Space applications must consider the effect energetic particles (radiation) can have on electronic components. In particular, SEUs may alter the logic-state of any static memory element (latch, flip flop, or RAM cell) or cause transient pulses in combinatorial logic paths. Since the user-programmed functionality of an FPGA depends on the data stored in millions of configuration latches within the device, an SEU in the configuration memory array might have adverse effects on the expected functionality of the user implemented design. Similarly, Single Event Transients (SETs) have a high probability for recognition at flip flop inputs where, if registered, causes a soft-error in the user data.

Static upsets in the configuration memory are not necessarily synonymous with a functional error; however, soft-errors are by definition a functional error. Upsets might or might not have an effect on functionality. However, an accumulation of upsets in the configuration memory is eventually certain to lead to a functional failure. Design mitigation techniques, such as triple module redundancy, can harden functionality against SEUs and SETs, while the SEUs are corrected so that static-errors do not accumulate and soft-errors do not propagate.

Implementing triple redundant circuits in other technologies, such as ASICs, is traditionally limited to protecting only the flip flops of the user's design from SEU, because logic paths in between the flop-flops are typically hard-wired, non-reconfigurable gates. For such fixed logic technologies, this is adequate protection from SEUs, but can still leave the circuitry vulnerable to SETs. For a technology that is vulnerable to SETs, further protection can be achieved through full module redundancy.

Full module redundancy is the required implementation of TMR in FPGAs, because all the logic paths, not just the flip flops, are susceptible to SEUs. This means that three full copies of the base design will be implemented to protect circuit functionality from SEUs, as well as SETs. However, the method for constructing TMR circuitry for Virtex FPGAs provides the additional advantages of complete data retention and autonomous recovery.

The correct implementation of TMR circuitry within the Virtex architecture depends on the type of data structure to be mitigated. These data structures can be grouped into four different types: throughput logic, state-machine logic, I/O logic, and special features (block RAM, Delay-Locked Loops (DLLs), etc.).

Throughput Logic

Throughput logic is a logic module of any size or functionality, synchronous or asynchronous, where all the logic paths within travel from the inputs to the outputs of the module without ever forming a logic loop. In other words, the logic states within a throughput logic structure are never dependent on their previous states. For example, an ADDER, of any size, is a throughput

logic structure. Regardless of how many clock stages may, or may not, lie between the inputs and outputs of the adder, the output is always a function of the inputs only. An accumulator, however, is not a throughput logic structure because the output is fed back into the inputs of the embedded adder. An accumulator is an example of a state-machine logic structure.

State-Machine Logic

State-machine logic is any structure where a registered output (at any register stage within the module) is fed back into any prior stage within the module forming a registered logic loop. This structure is used in accumulators, counters, or any custom state-machines or state-sequencers, where the given state of the internal registers are dependent on their own previous state. For example, a counter is really an accumulator that accumulates by one on every valid clock edge, but it can also be seen as a state-machine. If its current state is one, then its next state will be two (i.e., if it is a binary counter as opposed to a grey-code or one-hot style). Thus, if its current state is incorrect, then its next state is likely to be incorrect also.

I/O Logic

In this context, I/O logic refers to the inputs and outputs of the FPGA design. The scope of this application note is currently restricted to standard discrete low voltage TTL (LVTTTL) and low voltage CMOS (LVCMOS) type individual inputs and outputs. The current techniques do not incorporate the use of bidirectional or differential Input Output Block (IOB) circuits.

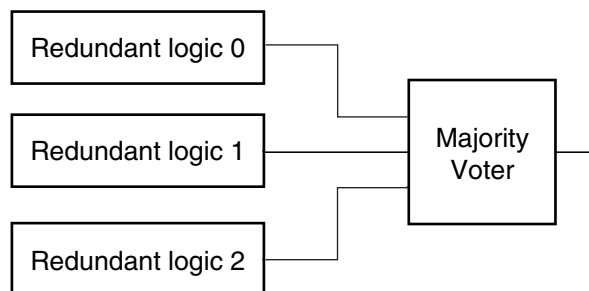
Special Features

The Virtex architecture provides a number of special features, such as block RAM, Look-Up Table (LUT)-shift-registers, LUT-RAM, and Delay-Locked Loops (DLLs), which require specialized methods for implementing effective redundancy. This application note currently covers techniques for the block RAM and DLLs. It should be noted that LUT-RAM and shift-register LUTs (SRL16) currently cannot be supported for Virtex and Virtex-E designs in applications that employ the use of Readback. See [Distributed RAM and Shift-Register LUTs](#), page 21.

Triple Redundancy and Voting

Majority Voters

The basic concept of triple redundancy is that a sensitive circuit can be hardened to SEUs by implementing three copies of the same circuit and performing a bit-wise “majority vote” on the output of the triplicate circuit. See [Figure 1](#).



XAPP197_01_030801

Figure 1: Triple Redundancy with Majority Vote

The circuit in question can be a mere flip flop or an entire logic design. The function of the majority voter is to output the logic value (“1” or “0”) that corresponds to at least two of its inputs. For example, if two or more of the voter’s three inputs are a “1,” then the output of the voter is a “1.” If the inputs of the voter are labeled A, B, and C, and the output V, respectively, then the boolean equation for the voter is: $V = AB + AC + BC$. The Truth-Table is shown in [Table 1](#).

Table 1: Majority Vote Truth-Table

A	B	C	V
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

The logic gate representation of the majority voter is shown in Figure 2.

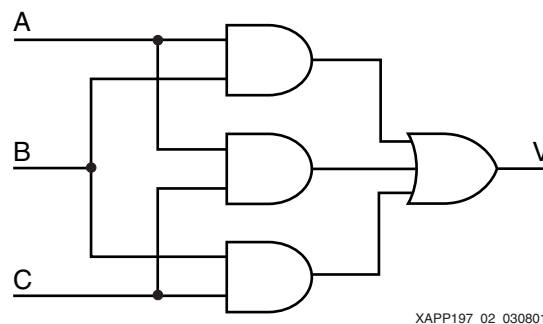


Figure 2: Majority Voter Circuit

Implementing Voters with 3-State Buffers

For designs constrained by available logic resources, the majority voters can be implemented using the Virtex internal 3-state buffers instead of Look-Up Tables (LUTs), which are used to implement all boolean functions in the user's design. The construction of a majority vote circuit using the Virtex library primitive BUFT is shown in Figure 3.

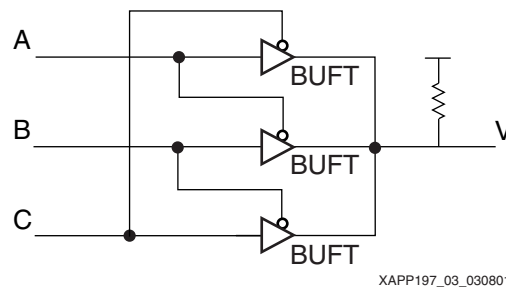


Figure 3: BUFT Style Majority Vote Circuit

The BUFT library primitive functions as an active low enabled 3-state buffer. In functional simulation, the circuit shown in Figure 3 drives an active Low on its output when all three of the inputs (A, B, and C) are Low. When all three of the inputs are High, the three BUFTs are disabled allowing the output to pull High. However, when only two inputs are Low, then a contentious and unknown state occurs in the simulator. See Figure 4. This means that the full functionality of a majority voter built this way cannot be simulated. However, a simulation of a

user's design should be free from SEUs. Therefore, all inputs to the voter should be the same allowing the data to pass through the voter. See [Figure 5](#).

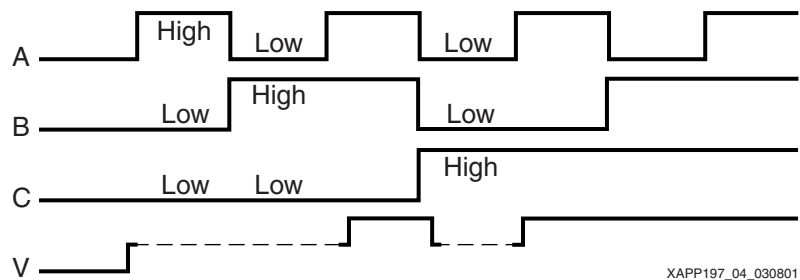


Figure 4: Functional Simulation of BUFT Voter

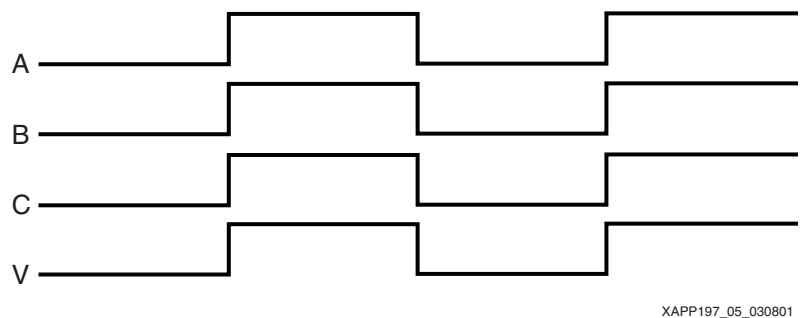


Figure 5: Proper Use of Voter in Functional Simulation

However, the hardware implementation takes on a very different form in the Virtex device. The contentious state described above does not exist and the voter actually works with the correct majority voting algorithm. See [Figure 6](#). This is due to the way in which the Virtex bussing logic is designed. [Figure 7](#) shows a representation of the horizontal bus logic in the Virtex architecture. This representation shows that there are two BUFTs per Configuration Logic Block (CLB) and four bus channels per row that are buffered through a junction every four columns. This structure allows segments of up to four BUFTs to be grouped into larger structures, while each BUFT can choose between two different segments. These selections are made by the Place And Route (PAR) software tools when implementing a user's design.

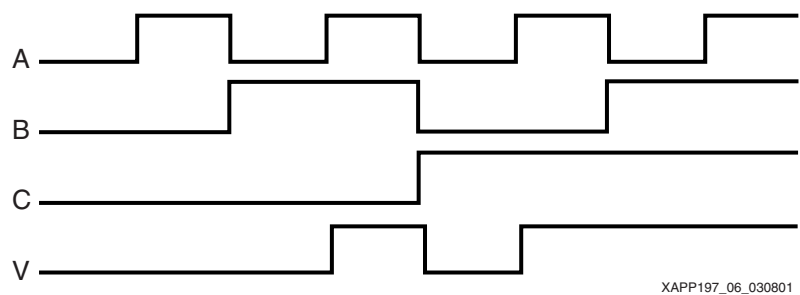
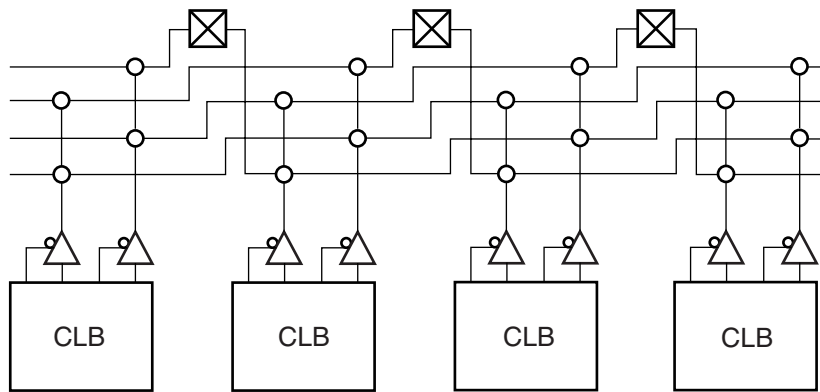


Figure 6: Actual Hardware Functionality of BUFT Voter



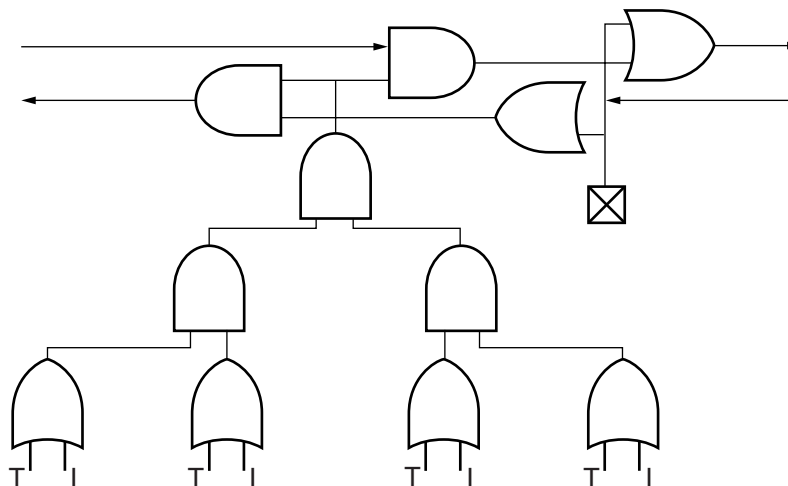
XAPP197_07_030801

Figure 7: Virtex Horizontal Bus Logic

The ability of these bus structures to implement majority vote circuits is because this architectural representation is merely a functional abstraction for the user. The actual circuit implementation is an entirely different form. Each junction actually represents a 4-bit look-ahead-carry bus segment shown in Figure 7. The four OR-gates with T and I inputs represent the inputs of four BUFTs, one per CLB. The junction has a bidirectional interface to other segment junctions to implement wider functions. In a majority voter, however, only three of the four inputs are needed. The fourth can be used by another junction. This allows all BUFTs in the device to be used without wasting any resources. From the circuit shown in Figure 7, the connections are made as described in Figure 3, resulting in the circuit shown in Figure 8. If distribution is applied to the equation in Table 1, the result is

$$AB + AC + BC = (A + B)(A + C)(B + C).$$

Therefore, Figure 8 is the same boolean function as the majority voter circuit shown in Figure 2.



XAPP197_08_030801

Figure 8: Virtex Bus Logic

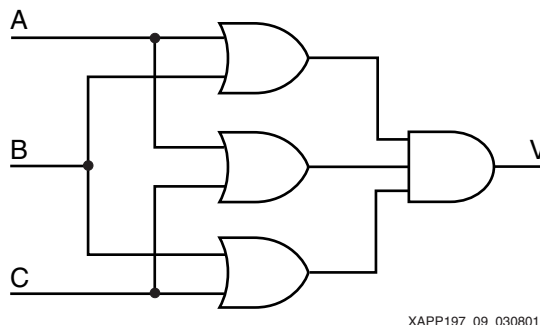


Figure 9: Majority Voter Circuit

Implementing Voters in Look-Up Tables

For FPGA designs that are not limited by available logic resources, but do require the fastest possible timing performance, building majority voters in the LUTs can provide a faster circuit implementation. The CLB LUTs are generally used to implement all combinatorial logic in the user’s design. Therefore, using LUTs to implement the voters allows the voter logic to be dissolved into general combinatorial logic with less propagation delay.

The type of voter used does not have any effect on the overall dynamic SEU cross-section, and both types can be used in the following examples. Therefore, from this point forward a majority voter is referred to as a TRV, and the type of voter (BUFT or LUT) intended for use is arbitrary.

Implementing TMR for Throughput Logic Structures

Logic Replication and Voting

To implement TMR for a throughput logic structure (see **Throughput Logic**, page 1), simply create three copies of the base module. This will also create three versions of each input and output. This will also be true for any logic structure, because the TMR methodology for FPGAs requires that the logic paths carry triple redundancy throughout the design to avoid a single point of failure.

Throughput logic does not typically represent an entire module by itself. A logic module most often contains various state-machines with combinatorial logic paths in between. However, implementing TMR can be simplified by creating hierarchical boundaries around individual state-machine logic structures, so that the TMR of one state-machine can be implemented as one module and the combinatorial logic that connects it to the next state-machine or I/O structure can be simply replicated.

Figure 10 shows a portion of an arbitrary base design where the output of a counter is decoded to trigger an event for another state-machine. The dashed-line boxes represent how the design should be broken into separate hierarchical modules.

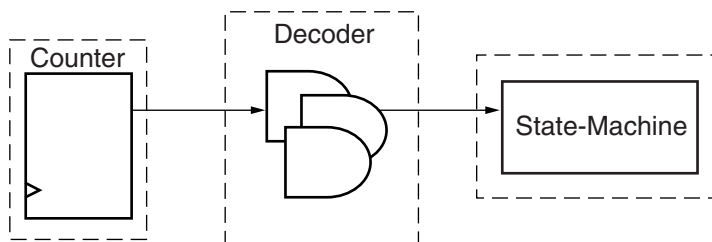
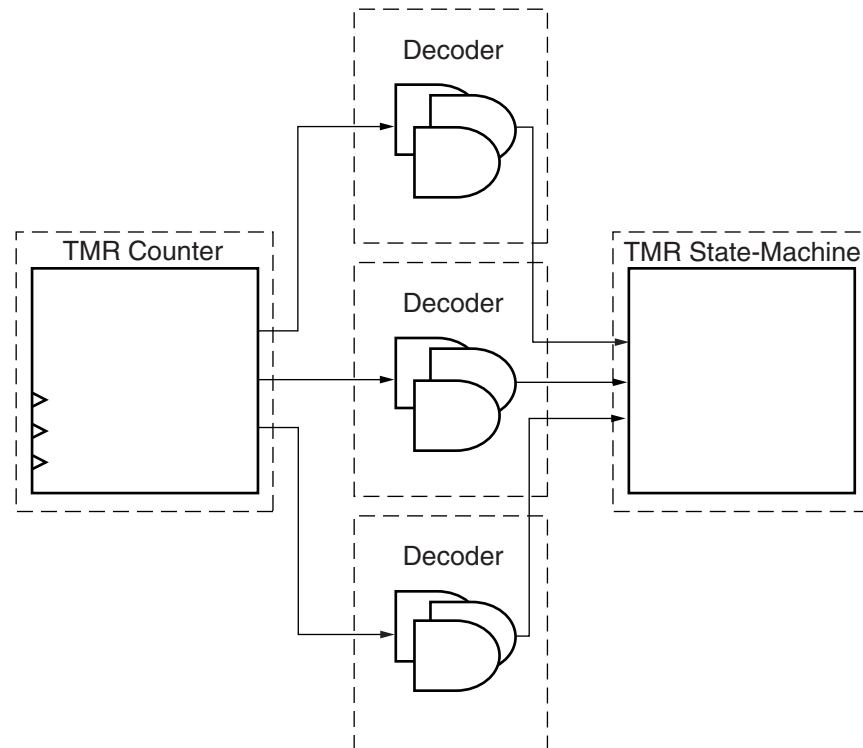


Figure 10: Arbitrary Logic Path

In **Figure 11** the TMR versions of the state-machines were created at a lower hierarchical level, while the combinatorial logic paths in between are simply replicated. No “voting” is required on these paths because they are voted inside the state-machines.



XAPP197_11_030801

Figure 11: TMR Logic Path

In **Figure 11**, the decoder logic is an arbitrary choice for a throughput logic structure. The decoder could have been absorbed into one of the two state-machines. However, the logic structure chosen for this example could just as well have been a large adder or multiplexer, or pipelined with hundreds or even thousands of register stages, anything large enough to make the necessity for hierarchical partitioning more obvious.

The point to this example is that the logic paths between the two state-machines are not state dependent. Any soft errors within the logic paths simply propagate through and do not get caught in any logic loops. The only purpose for the redundant logic paths is to carry the triple redundant output of the previous module to the triple redundant inputs of the next module without creating a single-point-of-failure. Since the three replicant modules do not need to be voted, there is no need for any interconnection between the three redundant legs. However, the exact opposite is true for state-machine logic.

Implementing TMR for State-Machines

The Simple State-Machine

Since state-machine logic is by definition state dependent, it is imperative that the TMR voting is performed internally rather than externally to such a logic module. A very simple, yet perfect example of this is a one-bit counter shown in [Figure 12](#).

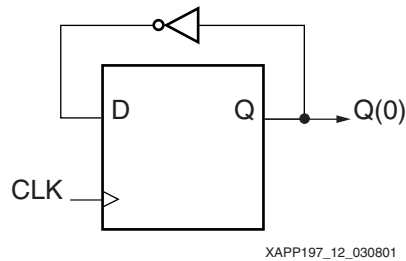


Figure 12: One-Bit Counter

The one-bit counter will load the opposite value of its previous state on every rising clock edge. Though the output is merely an alternating “1” and “0,” it is likely that there are other circuits that rely on a correct synchronization of the one-bit counter. [Figure 13](#) shows the results an SEU can have on that synchronization.

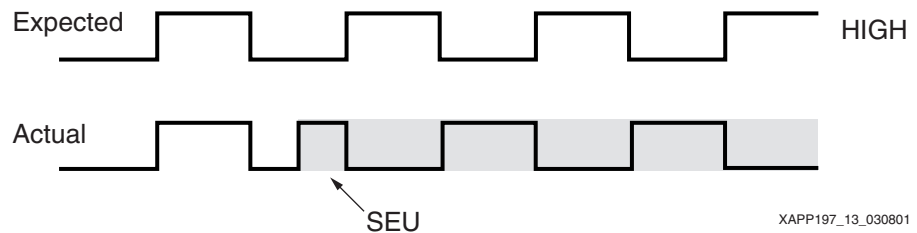


Figure 13: One-Bit Counter Output with SEU

The counter is now locked in an erroneous state because it is out of sequence and will remain that way until a master or host system resets the counter. If this circuit is replicated with three redundant versions and its vote on its output, as shown in [Figure 14](#), then the voted output will mask out the erroneous redundant leg.

This circuit generates a correct output in spite of an SEU. However, one logic leg continues to be out of sequence with the others. If another redundant leg is affected by an SEU, then the voted output also becomes permanently incorrect, as shown in [Figure 15](#). With two logic legs incorrect, the voted value is also incorrect and will not recover until the entire circuit is reset.

In order for the circuit to recover autonomously, i.e., without a host reset, each logic leg needs to incorporate the voted value into its logic feedback path. [Figure 16](#) shows this same circuit but using triple redundant voters in each of the feedback paths.

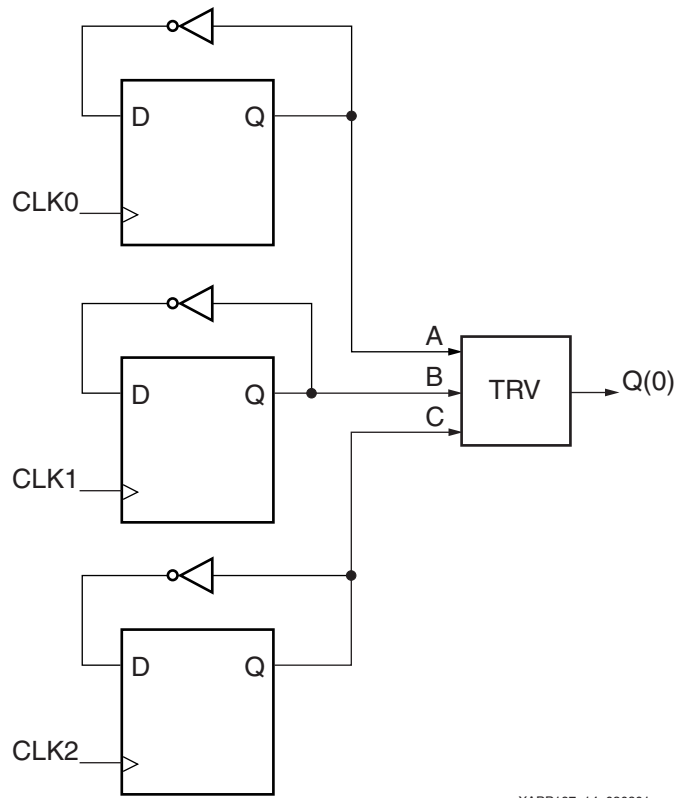


Figure 14: TMR One-Bit Counter

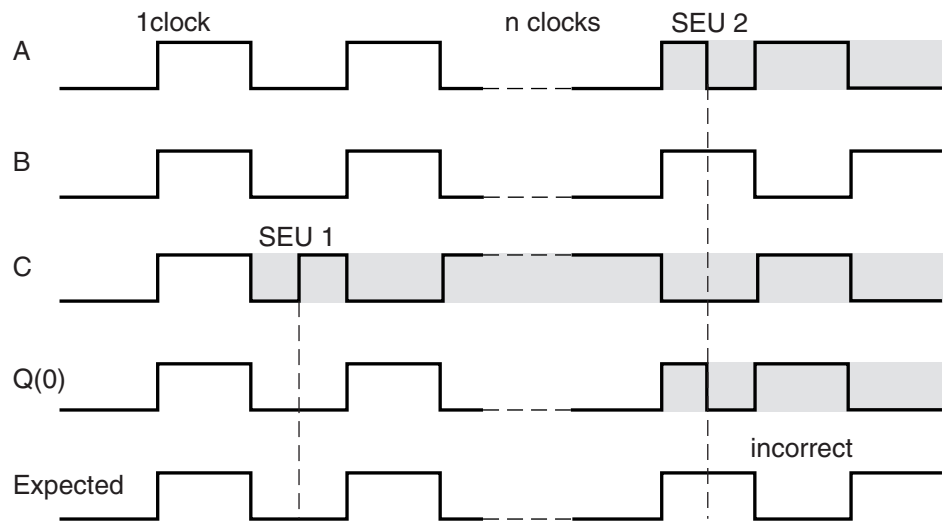
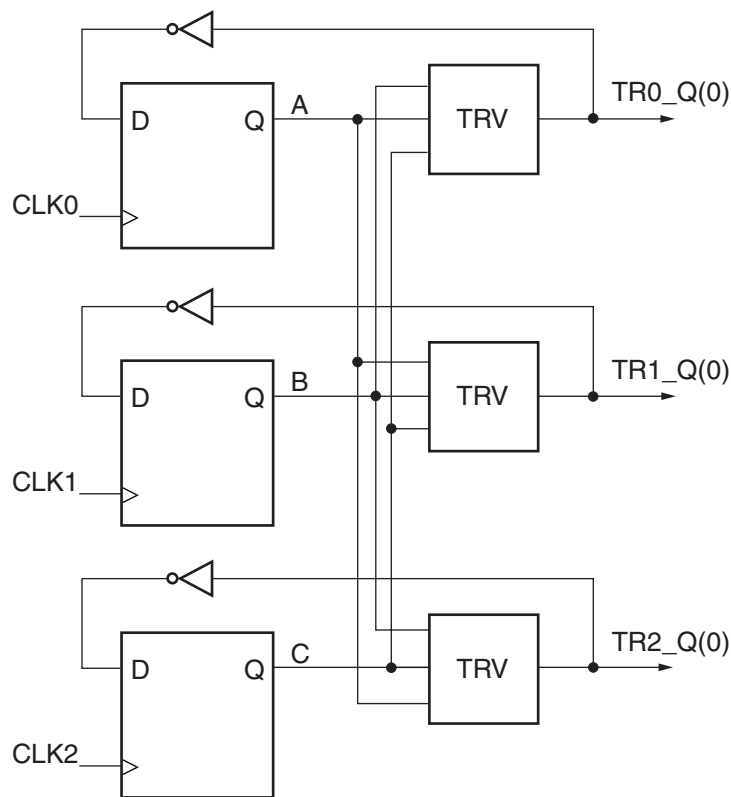


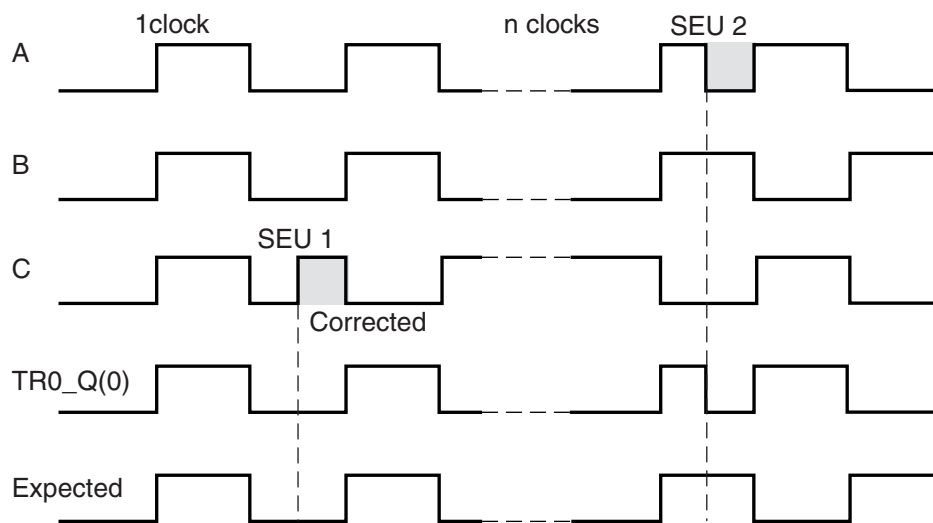
Figure 15: Single Voter TMR Counter with Sequential SEUs



x197_16_031201

Figure 16: Triple Voted Path TMR One-Bit Counter

As shown in Figure 17, soft errors get filtered out of the circuit on each clock step, allowing the circuit to recover before another SEU strikes.



XAPP197_17_031201

Figure 17: Triple Voted Path TMR Counter with Sequential SEUs

Therefore, the basic concept of constructing the TMR version of a state-machine is to triple all circuits and insert a majority voter into every registered loop or feed-back path. The use of three redundant majority voters eliminates these as single points of failure and provides the triple logic path outputs which get connected to the triple redundant inputs of the next module.

Since the redundant legs of the state-machine logic require cross-communications between the embedded voters, the TMR construction is best accomplished within a single module rather than separated into three. This is why the design is simplified by sinking state-machines into a sub-hierarchical boundary and excluding excessive throughput logic from being included within the boundary.

Ultimately, these triple redundant logic paths must be brought back together to form a single mitigated path which does not generate a single point of failure. This is accomplished within the TMR I/O circuitry.

Implementing TMR for I/O Logic

TMR Inputs

The primary purpose for using a TMR design methodology is to remove all single points of failure from the design. This begins with the FPGA inputs. If a single input was connected to all three redundant logic legs within the FPGA, then a failure at that input would cause these errors to propagate through all the redundancies, and thus the error would not be mitigated. Therefore, each redundant leg of the design that uses FPGA inputs should have its own set of inputs (Figure 18). Thus, if one input suffers a failure, it will only affect one redundancy.

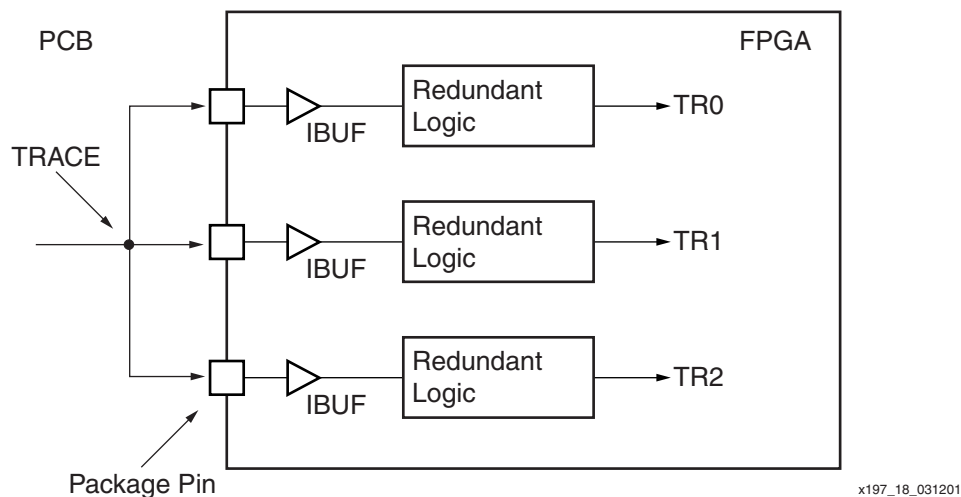


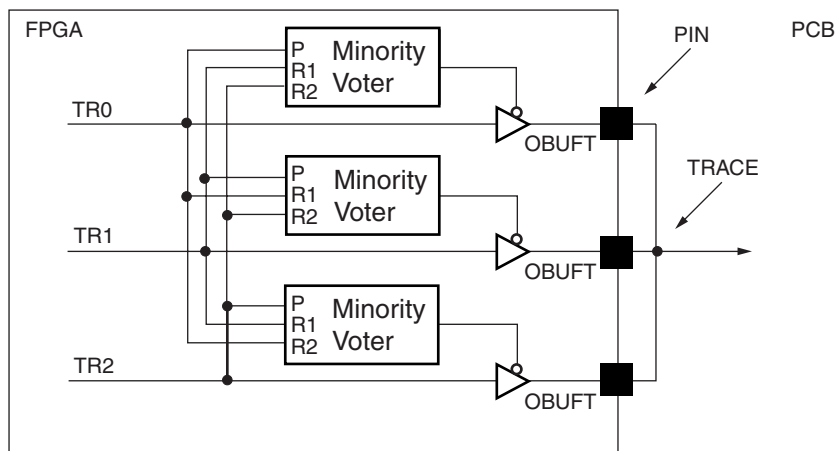
Figure 18: Triple Redundant FPGA Inputs

This does, however, place limitations on certain resources. Tripling inputs and outputs causes a severe reduction in available I/O. This method should be used whenever there are enough resources to realize all the inputs and outputs of the design in triplicate. However, if the basic design requires more than one third the total available I/O on the device, then the user needs to consider the trade-offs of either not tripling all I/Os or partitioning the design into multiple devices.

TMR Outputs

The outputs are the key to the overall TMR strategy. Since the full triple module redundancy generates every logic path in triplicate, there must ultimately be a method for bringing these triple logic paths back to a single path that does not create a single point of failure. This can be accomplished with TMR outputs.

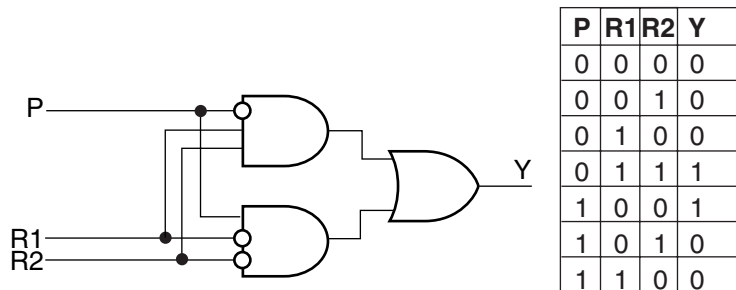
A TMR output is constructed using the OBUFT library primitives as shown in Figure 19. Each redundant logic path exiting the FPGA on an output does so through an OBUFT. The “enable” (T pin) of each OBUFT is controlled by a “minority voter” circuit. The minority voter indicates whether the path in question (primary path) agrees with either of the two redundant paths. If the primary path agrees with at least one of the redundant paths, then the primary path is considered to be part of the majority. If the primary path disagrees with both redundant paths, then the primary path is in the minority.



XAPP197_19_031201

Figure 19: Minority Voted TMR FPGA Outputs

The minority voter is shown in Figure 20. If the primary path is part of the majority, then the minority voter will enable the corresponding (active Low) OBUFT allowing the data on its primary path to be driven out through the OBUFT and onto the Pad-Pin. If the primary path is not a part of the majority, then the OBUFT is disabled placing its output in a high-impedance state allowing the redundant outputs to drive the correct data.



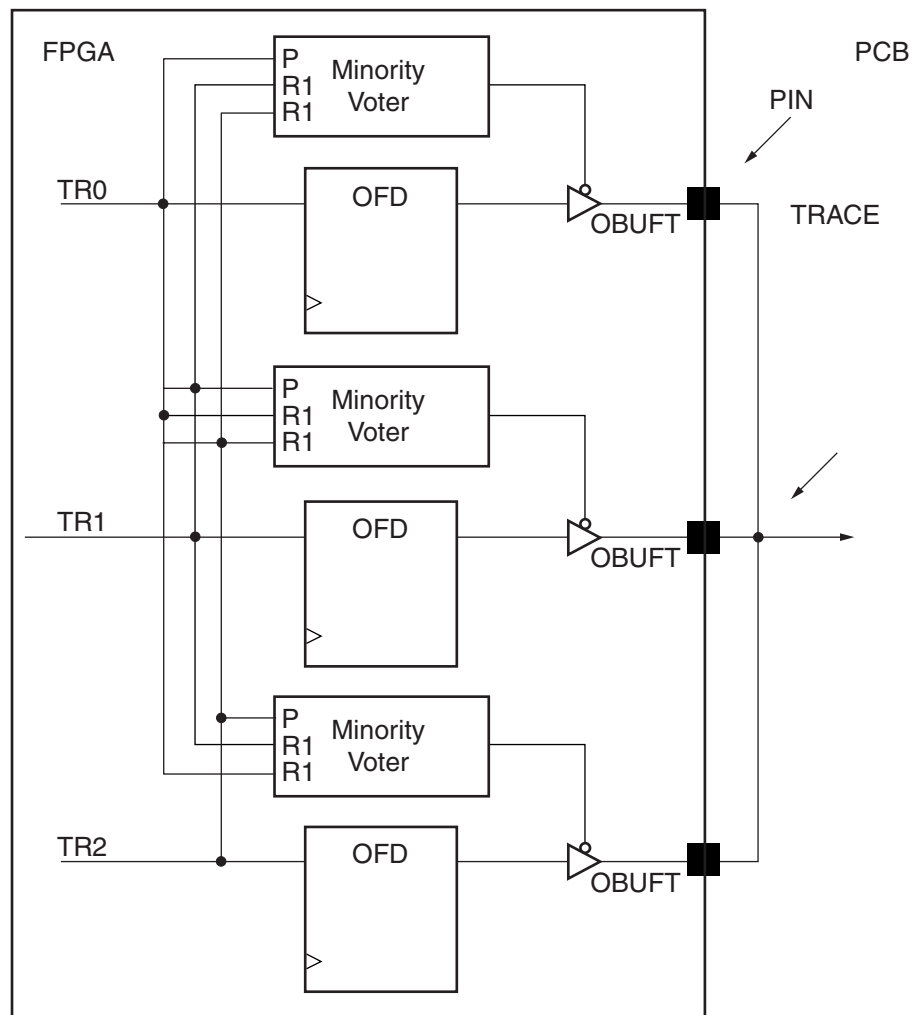
XAPP197_20_031201

Figure 20: Minority Voter Circuit

Externally from the FPGA, the three outputs are hardwired together on the circuit board. This structure does not cause any contentious states because only paths that agree with each other are actively driven. This method also has the added benefit of doubling and/or tripling the sink and source current capabilities of the output from the perspective of other components on the board that are connected to this board trace. But the primary advantage to this method is that no external devices are needed to complete the triple redundant voting, as would be the case when using triple redundant FPGAs instead of internal redundancy within a single FPGA.

IOB Registers

The IOBs in the Virtex architecture also contain dedicated separate input and output flip flops. While there are no restrictions for using the input registers, the use of output registers (OFD) in conjunction with the TMR output method, shown in Figure 19, should be avoided. The minority voters can only sample signals *before* they pass to the IOBs. Since the IOB registers exist inside the IOB, their usage would have to bypass the voters (Figure 21).



XAPP197_21_031201

Figure 21: Incorrect TMR FPGA Outputs

If this output structure is used, then an SEU to one of these registers could place the outputs in contention. The circuit shown in Figure 21 is illegal because the signals are registered after they are voted. Optimization software, such as the Xilinx Map tool, as well as synthesis tools, can be selected to “pack registers” into IOBs when possible. However, as long as the last register in the path comes before the minority voters, then the voters (which are implemented in CLB logic) will keep the registers from being mapped into the IOB.

Inter-FPGA Communications

For TMR output signals that simply pass from one FPGA to an adjacent FPGA, the TMR output voting structure, shown in Figure 19, can be omitted and the three redundant outputs can be separately traced to the adjacent FPGA. One advantage for this would be no restrictions on the use of IOB output registers, thus, allowing better clock-to-output performance. Another advantage is that other I/O standards, such as GTL or LVDS can more easily be used for high-speed chip-to-chip communication.

Special Architecture Features

While the majority of any logic design can be realized in Look-Up Tables, flip flops, and routing, there are other special features specific to the Virtex architecture that allow for more efficient and higher performance implementations. These features include block RAM, LUT RAM, Shift-registers, Arithmetics, and clock DLLs. While SEU mitigation methods are still being developed and tested, the following are recommendations for some of these features.

Block RAM

The Virtex block RAMs are large blocks of static memory (4K bits each) that are true dual port and fully synchronous. True dual port means that both ports have read and write capability. The data content of the block RAM can be accessed through the configuration (SelectMAP) port, but this is an inadvisable method for Error Detection and Correction (EDAC).

When the block RAM content is accessed through the configuration port, this process disables access to the block RAM from the user logic, thus disrupting the user's design operation. Therefore, it is recommended that any necessary EDAC for the block RAM content should be incorporated into the user specified design.

Simple Redundancy

One method is to not include any error detection and correction, but instead use triple redundant block RAMs and majority voters on the outputs. This will be sufficient for an application that is likely to write new data to all memory addresses within the time that upsets could be expected in similar addresses in the redundant blocks.

This method relies on the statistical upset rate and is not the most safe and secure. However, it does allow for maximum feature usability, because no additional overhead is introduced to refresh the memory blocks.

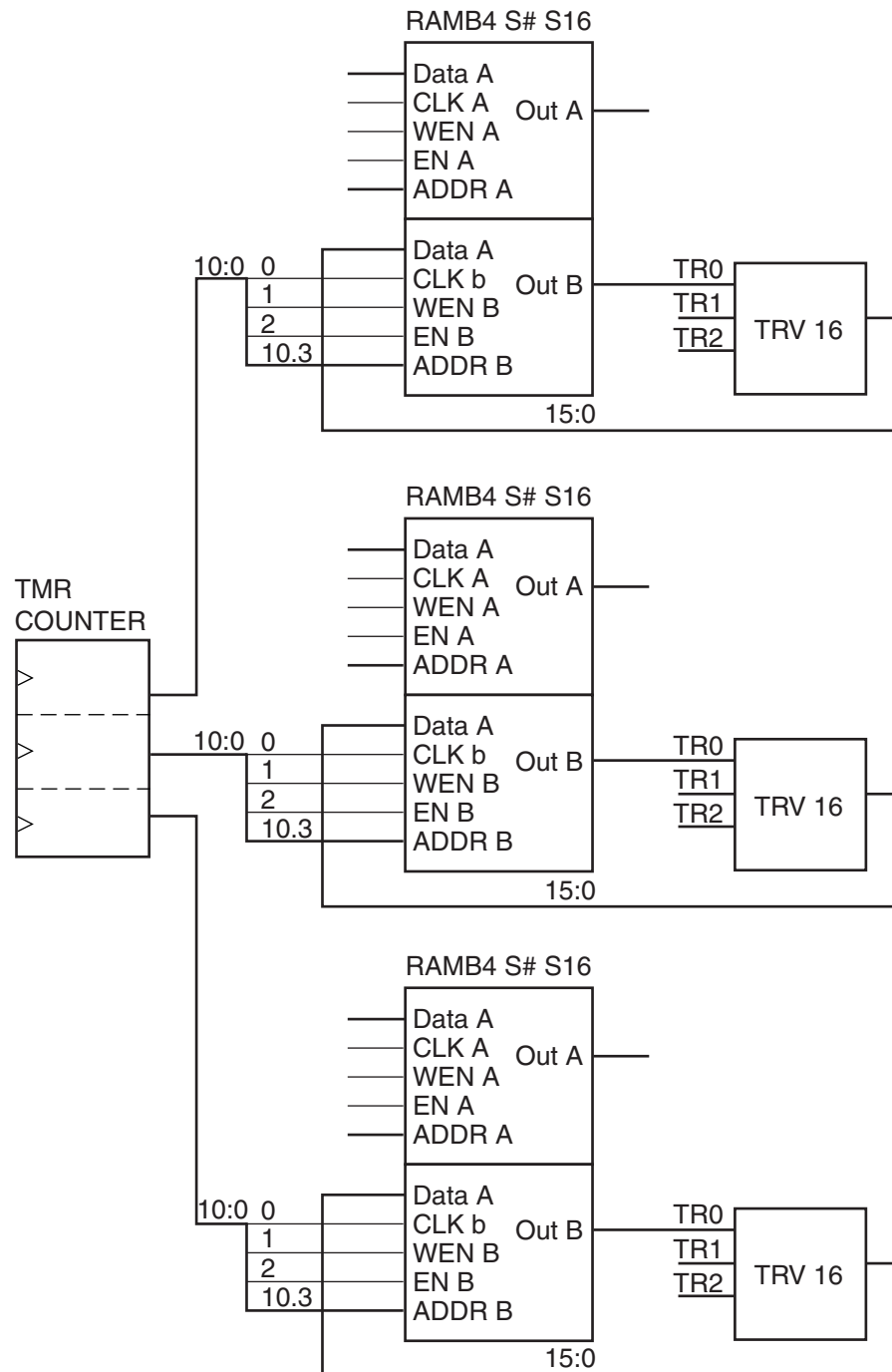
Redundancy and Refresh

A more reliable method is to constantly refresh the block RAM contents. Since these are dual port memories, one of the ports can be dedicated to error detection and correction. But this also means that the block RAMs can only be used as single port memories by the rest of the user logic.

To refresh the memory contents, a counter can be used to cycle through the memory addresses incrementing the address once every four clock cycles. See [Figure 22](#). The Bit[0] index of the counter output drives the CLK of port B. Bit[1] and Bit[2] indices drive the WEN and EN, respectively. For each address, the data content is voted and the majority vote value written back into the cells.

In this example, the data width of port B is set to its maximum value of 16. This reduces the address width, and thus the counter size as well, to its minimum value of 8. However, the data width of Port A can be set independently of Port B and used in the application design.

One should also realize that the outputs of Port A do not need to pass through majority voters because they are already triple redundant. Voting occurs at the next state-machine or I/O stage in the design.



XAPP197_22_031201

Figure 22: TMR Block RAM with Refresh

Data Encryption

Another method is to encode the data before writing it to memory and then decode that data when reading it from memory. EDAC algorithms, such as “Hamming,” have been used in space applications that utilize on-board memories.

Data is encrypted before it is written to memory and then decoded after a memory read. The encoding algorithm allows the correct data to be extracted from partially corrupted data.

Clock Management

The Virtex architecture has four clock buffers and four DLLs for implementing clock resources within the design. This is somewhat limiting for a TMR design because this will only allow for a single TMR clock domain. However, high-fanout low-skew clock trees can easily be implemented in the Virtex routing without using these special clock resources. In fact, any I/O can serve as a secondary clock input allowing for multiple TMR clock domains.

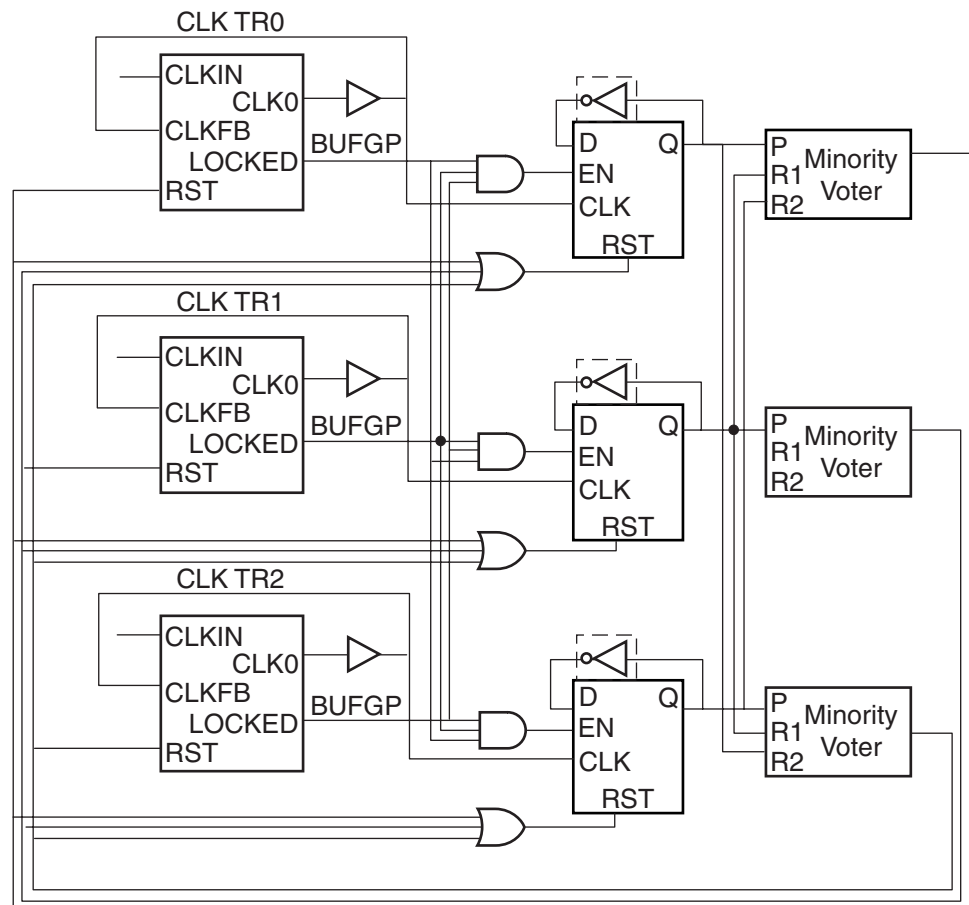
Clock Buffers

A design that only uses a single system clock can use the primary clock buffers (BUFGP) to make a single TMR clock domain. This is in fact three clock domains that are externally driven by the same clock. Each of the three BUFGP global clock buffers fanout to one complete redundant leg of the whole FPGA design that may exist in this domain. Additional clock domains have to use other resources.

Clock DLLs

The DLLs can be used in conjunction with the BUFGPs to resynchronize the clock signal to its own path skew or an external reference to decrease clock-to-output delays. However, an SEU in the DLL circuitry can have the effect of unsynchronizing the DLL. This can result in jitter or complete loss of the output clock signal.

Though the DLLs have an output signal “LOCKED” that is used to display the status of the DLL synchronization during normal operation, this output cannot be used as a reliable SEU detection. When a DLL is upset, it must be reset in order to resynchronize. This can be accomplished with a small SEU detection circuit shown in **Figure 23**.



XAPP197_23_031201

Figure 23: TMR Clock DLLs with Self-Test

Three one-bit counters are used to demonstrate that all three clocks are running and synchronized. Each counter is clocked from one of the redundant clock outputs. The Enable to the counters is not asserted until all three DLLs are synchronized and have asserted their LOCKED outputs. When one of the three register outputs is not in agreement with the majority, its associated minority voter resets the proper DLL, as well as the three registers. The three registers will not begin clocking again until all three DLLs are in lock.

Each DLL requires a clock input on the CLKIN pin. This input can only come from a dedicated clock input pad (GCLK) through an input clock buffer (IBUFG: not shown). The clock outputs are CLK_TR0, CLK_TR1, and CLK_TR2. These represent the internal TMR clock domains. The DLL also has other simultaneous outputs that provide a 90, 180, and 270 phase shift, a 2x clock multiplication, and a user specified clock division output.

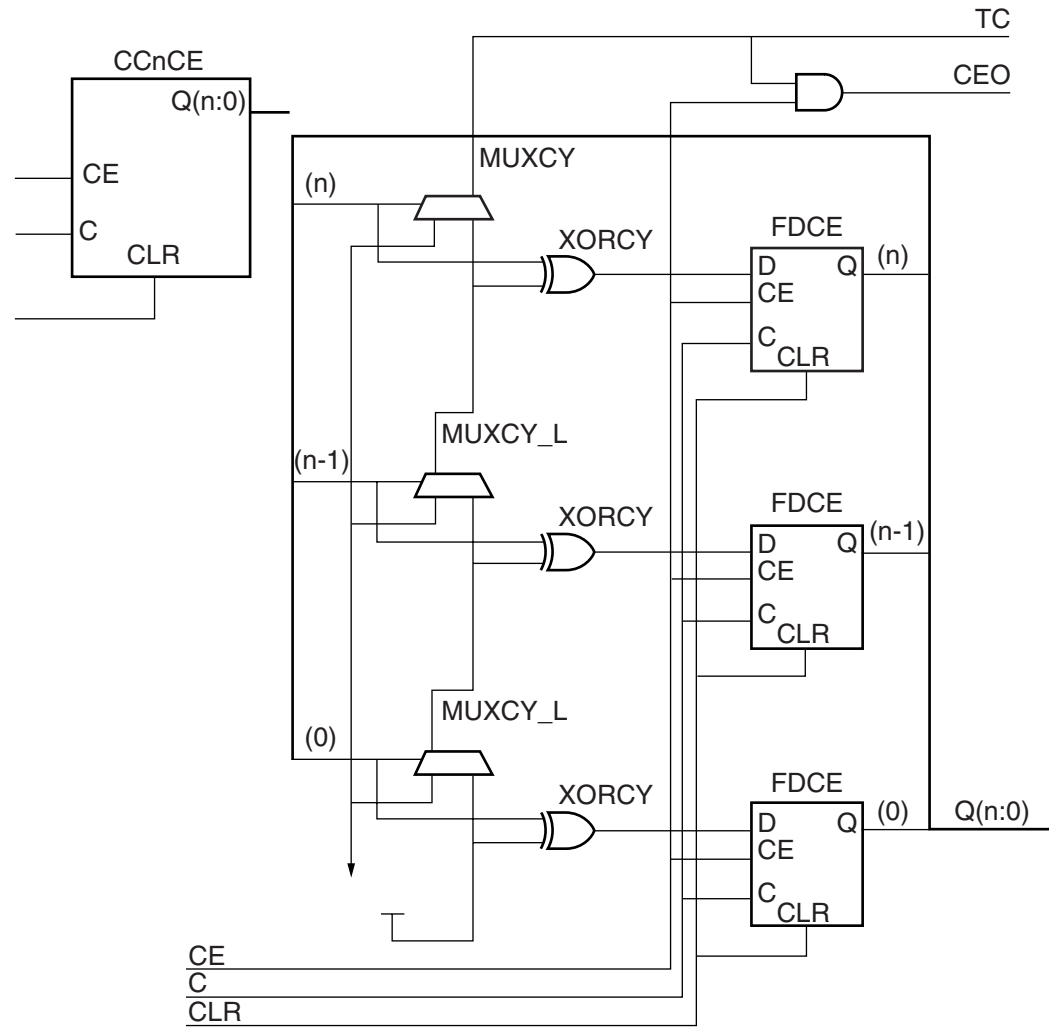
Arithmetic Carry Chains

Arithmetics, such as counters and adders, are most efficiently implemented using the carry-chains imbedded within the CLBs. The typical user is not likely to build carry-chain structures at the primitive level, but will likely instantiate library macros that utilize these features or infer their usage when synthesizing an HDL-based design. However, neither the standard Xilinx library nor synthesis libraries take TMR design methods into account.

Schematic Designs

The schematic designer can copy and modify a Xilinx macro from the EDA specific schematic library to make a custom TMR version of it. The carry-chain implementation for a basic counter is shown in [Figure 24](#). The first stage (bottom) initializes the carry chain. The final stage (top) uses a MUXCY instead of a MUXCY_L, so that the carry-out signal can leave the carry-chain logic and exit the CLB to be used as a terminal count (TC). Any number of identical stages can be placed in the middle to create any counter size (limited by the number of CLBs per column for a particular FPGA family member).

Symbol



x197_24_031901

Figure 24: Standard Carry Chain Counter

The schematic in Figure 24 has an obvious registered loop. To create a TMR version of this, the internal loop must be broken so that majority voters can be inserted. This is shown in Figure 25.

Symbol

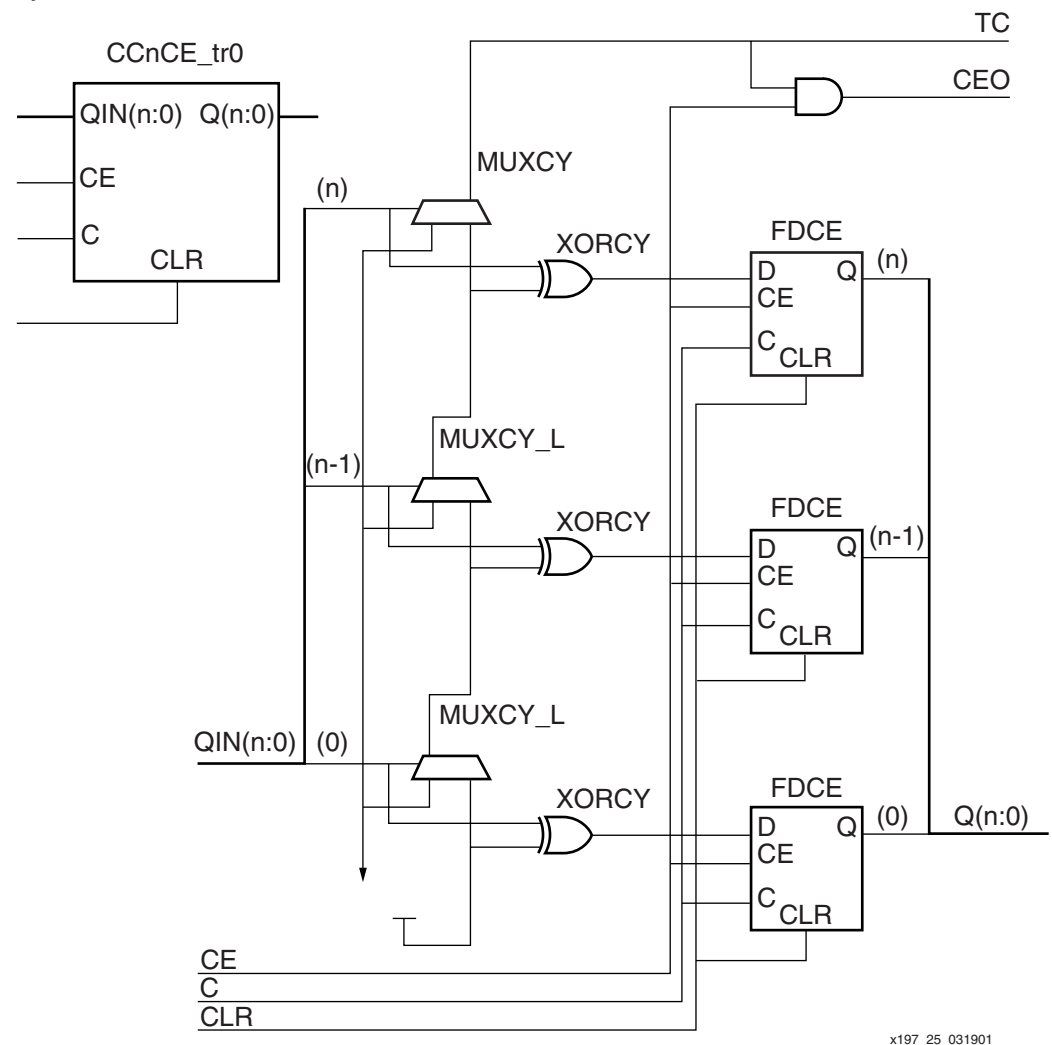
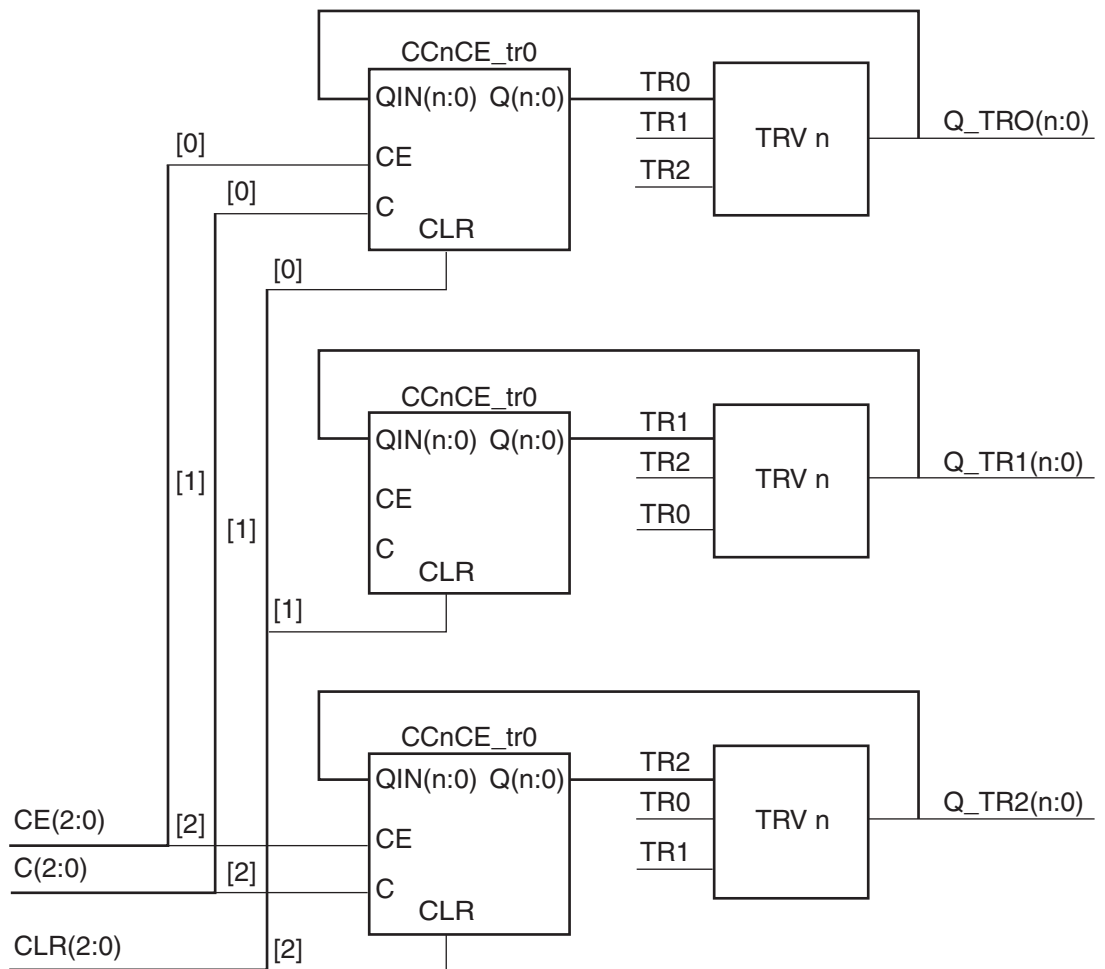


Figure 25: Carry Chain Counter Modified for TMR

As we see in Figure 25, the feedback path has been severed and a new input ($QIN(n:0)$) has been added to the symbol. Now three of these modified counters can be used to construct a TMR counter, shown in Figure 26.

Notes:

1. The work here is not yet complete. There is also an additional issue with regards to the use of V_{CC} and GNDs, such as those shown in Figure 24 and Figure 25. See "V_{CC} and GND Extraction" on page 21.



XAPP197_26_031201

Figure 26: TMR Counter: CCnCE_TMR

As shown in Figure 26, three instances of the new macro can be combined with the majority voters to create a TMR counter macro. The clock enable (CE), clock (C), and clear (CLR) are triple redundant and separate for each redundant leg. However, the separate redundant outputs are each a majority vote of all three counters, each providing a separate feedback path for the counters. Therefore, if one counter falls out of step as a result of an SEU, it corrects itself on the first clock cycle after the SEU is fixed (assuming that the likeliest place for an SEU is in the configuration memory. An SEU in one of the flip flops is immediately corrected by virtue of the TMR circuitry).

Designs in VHDL

The HDL designer typically infers a counter function similar to the following VHDL example.:

```

Counter: process(CLK, CLR)
Begin
  If (RST='1') Then
    Q <= null; (others => 0);
  Elself CLK'event and CLK='1' then
    if (CE='1') then
      Q <= Q + 1;
    end if;
  End If;
End Counter;

```

Although this coding style can easily be adapted to infer a similar function as that demonstrated in Figure 26, this would leave an unresolved issue concerning the use of V_{CC} and GND components in the design. See "V_{CC} and GND Extraction" on page 21.

The preferred method of implementing any carry-chain logic structures is to instantiate a macro from the XQVR_SYN library. See "Other Examples in VHDL" on page 34.

Distributed RAM and Shift-Register LUTs

LUTs may be used as small blocks of distributed RAM elements (e.g., RAMS16x1) or as dynamically addressable shift registers (e.g., SRL16) in the user's design. When a LUT is used for this type of operation, the user's data content is dynamically stored and manipulated in configuration memory cells. This poses a problem for an application that intends to use Readback and/or partial reconfiguration.

Since space applications depend on the use of readback and partial reconfiguration for SEU detection and correction in the configuration memory, it is highly recommended that users do not use LUTs in this way, as it can cause data corruption in the configuration memory.

It is recommended to use the block RAM memories for all RAM functions and flip flops for shift-registers.

V_{CC} and GND Extraction

Persistent Errors

A typical FPGA design is implemented with signals that were resolved to a logic constant (V_{CC} or GND), but which could not be entirely optimized out of the design. When V_{CC} s and/or GNDs are implemented by the PAR tools, they are implemented in a way that maximizes device resource utilization. This is accomplished by utilizing "Keeper" circuits that exist at the input pins of all CLBs and IOBs.

Keepers lie in series with routing channels and logic block input pins. When the routing channel carries an active signal, the keeper is transparent. But when the channel is unused, the keeper keeps its last known value, which was determined when the device was initially powered-up or re-initialized by activating the FPGA input PROG.

When a logic element (e.g., flip flop) inside a logic block (CLB or IOB) requires a logical constant, such as a V_{CC} or GND, this logical constant can be obtained from the keeper circuit of an unused pin of the logic block. Its polarity can be selected by programmable inversion within the logic block.

An SEU can upset or alter the state of a keeper circuit either by direct ionization, or indirectly by momentarily connecting an active routing channel to the input of the keeper. In either case, the result is a functional disturbance that cannot be detected by readback nor corrected by partial reconfiguration. Therefore, this type of error is known as a "Persistent Error," and it can only be corrected by completely re-initializing the FPGA.

This sensitivity can be eliminated by removing the functional dependencies on V_{CC} s and GNDs from the user's design.

Where to Find V_{CC} and GND

V_{CC} and GND exist in the Xilinx library for all product families as primitive elements. These should not be used in designs that are concerned with SEU mitigation. The HDL designer may inadvertently cause their usage by assigning a constant value to a signal or port, by not defining a clear usage for inputs of a primitive element, or by inferring arithmetics that are ultimately implemented in carry-chain logic. But V_{CC} and GND also exist within Xilinx specific schematic and synthesis library macros.

Flip Flops

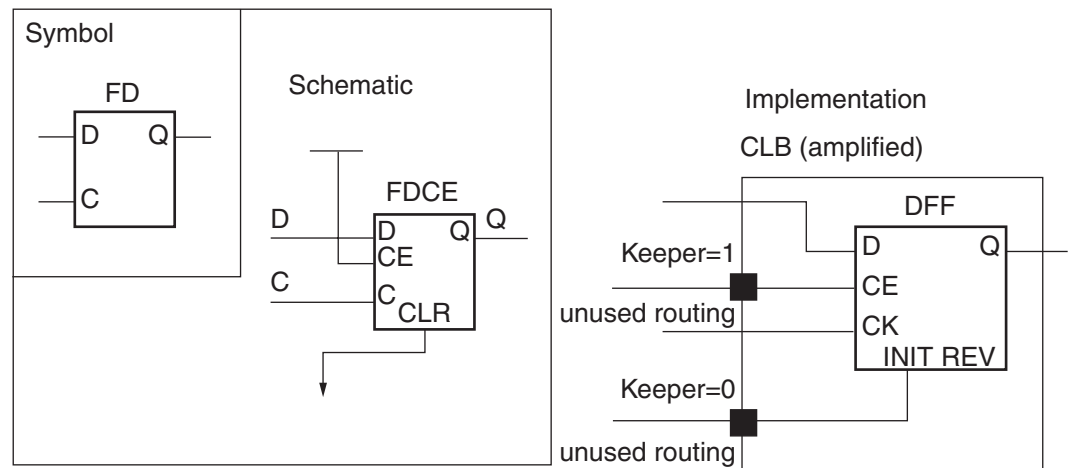
The primitive Virtex CLB flip flop has a Clock Enable (CE), as well as INIT and REV inputs (an IOB register has a CE and INIT, but not a REV). The INIT and REV are used to implement synchronous and/or asynchronous SET and RESET functions for the register. The CE and the

INIT must be driven for the register to operate, even if this connection is not specified in the user's design.

For example, if the schematic user instantiates an FD (D-type flip flop) in their design, they have in fact instantiated a library macro that implements an FDCE with its CE pin tied to V_{CC} and its CLR pin tied to GND (the CLR eventually becomes the INIT).

Schematic designers should be careful to examine the primitive implementation of all library macros that are likely to contain registers, before using them in their design. Even if the macro provides clock enable and reset pins at the top level, the primitive implementation might be different than expected.

Similarly, if a VHDL user describes a synchronous process without specifying a clock-enable or initialization function, the synthesis tool implements this function by using primitives and tying all unused pins to the correct logical constant, thus, creating V_{CC} s and GNDs. The following VHDL example creates the same problem as that shown in Figure 27.



x197_27_031901

Figure 27: Primitive Implementation of a D-type Flip Flop

```

register: process(CLK)
Begin
  If CLK'event and CLK='1' then
    Q <= D;
  End If;
End register;

```

Synthesis designers should always include synchronous clock enables and either synchronous and/or asynchronous initialization conditions for all synchronous processes and component instantiations:

```

register: process(CLK, CLR)
Begin
  If (CLR='1') Then
    Q <= 0;
  Elsif CLK'event and CLK='1' then
    if (CE='1') then
      Q <= D;
    end if;
  End If;
End register;

```

Arithmetic Carry Chains

Looking again at [Figure 25](#), notice that the carry-chain is initialized with a V_{CC} and GND. One simple method for the schematic designer to remove these is shown in [Figure 28](#). Since the CE signal must be asserted High for the counter to increment, the V_{CC} can be replaced with a connection to this signal. Similarly, since the CLR signal must be asserted Low for the counter to increment, the GND can be replaced with a connection to the CLR signal.

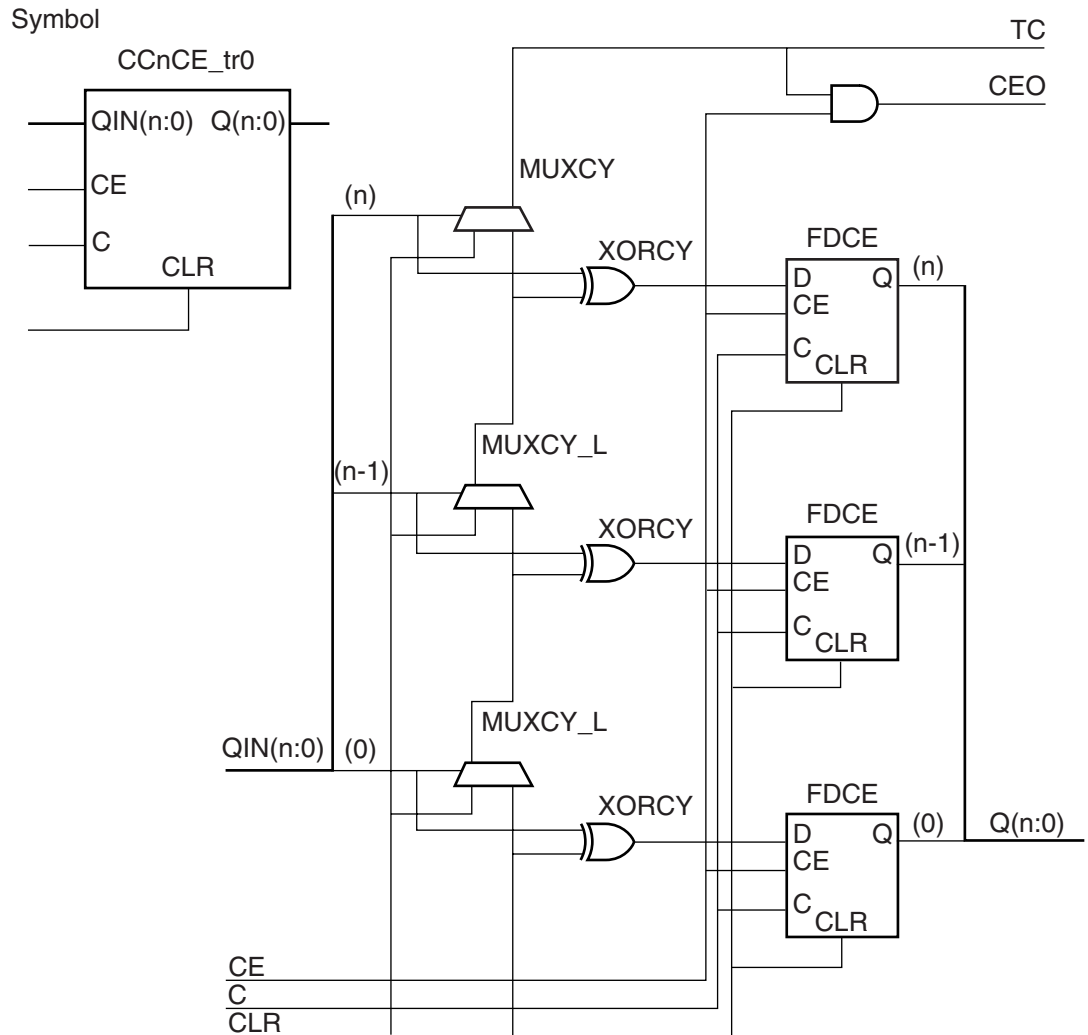


Figure 28: Carry Chain Counter Modified for TMR with Power-Tie Downs

The only way for the synthesis designer to accomplish the same implementation is to literally instantiate their own primitive representation of the carry-chain logic or to use one of the provided macros in the XVRWARE library. See ["Other Examples in VHDL" on page 34](#). Inferring arithmetic functions through behavioral coding always generates implicit V_{CC} and GND usage in synthesis. While this might be a little bit inconvenient, it also has a significant impact on the expected SEU error rates for your design.

Power Tie Downs

Now all the V_{CC} and GNDs have been replaced with signal connections. If the users design did not already have an available Global Clock Enable or Global Reset signal, or cannot be adapted to create one, then a new active and routable signal must be created in the design to provide a connection point for the needed logical constants.

If any of these signals are left unconnected in the design, the optimization either leads to excessive logic deletion, or the re-insertion of V_{CC} and GNDs. One simple method to provide a unilateral V_{CC} and GND function is to dedicate an I/O pin (three for triple redundancy) to import an actual ground connection from the circuit board. This is demonstrated in Figure 29.

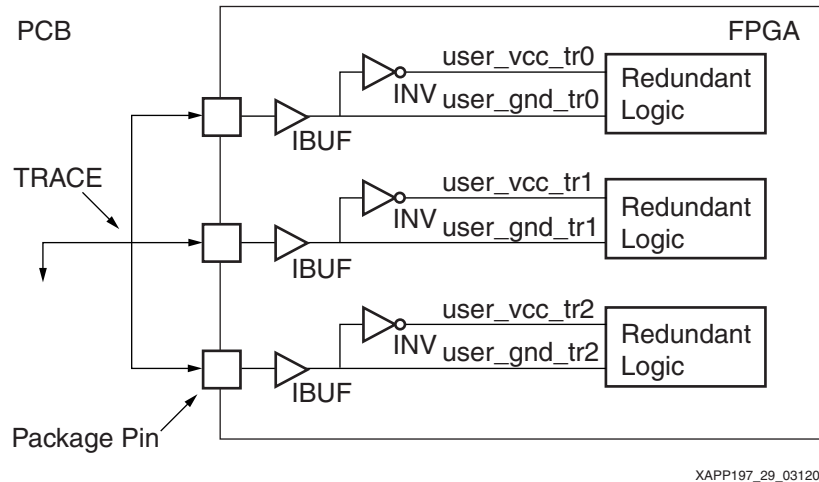


Figure 29: Triple Redundant Power Tie Downs

XVRWARE Macro Synthesis Library

Library Macros

The XVRWARE Synthesis library provides macros and synthesis examples for constructing TMR circuits in VHDL for the Virtex architecture. The XVRWARE library can be downloaded from the Xilinx site: [xapp197.zip](http://www.xilinx.com/xapp197.zip) and should be extracted in a user's area or directory separate from the Xilinx installation area. The first release of this library provides macros specifically designed for use with Synplicity's *Synplify* and *Synplify Pro*.

The XVRWARE macros are in the *xvrware/Synplicity/src* subdirectory. The *Synplicity/examples* subdirectory provides usage examples for the library macros, as well as general TMR methods for state-machines and power-ties. The following section lists the ports and function of each library macro.

Voters

TRV_LUT and TRV_BUFT

The TRV_LUT and TRV_BUFT macros are triple redundant voter implemented in either LUT or 3-state buffers (BUFT).

Table 2: TRV (LUT/BUFT) Port Definitions

Port Name	Direction	Type	Width
TR0	IN	Std_Logic	1
TR1	IN	Std_Logic	1
TR2	IN	Std_Logic	1
V	OUT	Std_Logic	1

Table 3: TRV Function Table

TR0	TR1	TR2	V
0	0	0	0
0	0	1	0
0	1	0	0

Table 3: TRV Function Table (Continued)

TR0	TR1	TR2	V
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

OUTPUT_TMR

The OUTPUT_TMR macro is for triple redundant LVCMOS outputs with path voting. Three separate OBUFT output drivers each are enabled by a separate minority vote. The triple outputs are designed to be connected together externally from the FPGA to a single board trace. Properly connected, this macro provides a three input, one output majority vote function for the purpose of combining three internal redundant signals to form a single voted external signal.

Table 4: OUTPUT_TMR Port Definitions

Port Name	Direction	Type	Width
TR0	IN	Std_Logic	1
TR1	IN	Std_Logic	1
TR2	IN	Std_Logic	1
V0	OUT	Std_Logic	1
V1	OUT	Std_Logic	1
V2	OUT	Std_Logic	1

Table 5: OUTPUT_TMR Function Table

TR0	TR1	TR2	V0	V1	V2	Wired
0	0	0	0	0	0	0
0	0	1	0	0	Z	0
0	1	0	0	Z	0	0
0	1	1	Z	1	1	1
1	0	0	Z	0	0	0
1	0	1	1	Z	1	1
1	1	0	1	1	Z	1
1	1	1	1	1	1	1

Counters

The TMR Counter macros are similar to the circuits shown in Figure 26 and Figure 28. The counter macros have the majority voting built into the carry-chain logic, thus making use of otherwise inaccessible LUTs. Additionally, the necessary V_{CC} and GND connections for carry-chain initialization are ported to USERVCC and USERGND, respectively. These can either be connected to the CLR and CE lines, as shown in Figure 28, or should be connected to global USERVCC and USERGND signals. Depending on which macro is used, this can also either be the PRE, R, or S port. The user **must** make this connection at the instantiation point in their design to activate indissoluble signals, such as an FPGA input.

CCwCE_TMR

The CCwCE_TMR macro is a parameterized triple redundant up counter with asynchronous clear and clock enable. It is functionally equivalent to the CC16CE Virtex library macro. The generic map property Width specifies the data width of the counter's output.

Table 6: CCwCE_TMR Port Definitions

Port Name	Direction	Type	Width
CLK	IN	Std_Logic_Vector	3
CLR	IN	Std_Logic_Vector	3
CE	IN	Std_Logic_Vector	3
USERGND	IN	Std_Logic_Vector	3
TC	OUT	Std_Logic_Vector	3
CEO	OUT	Std_Logic_Vector	3
Q_TR0	OUT	Std_Logic_Vector	WIDTH
Q_TR1	OUT	Std_Logic_Vector	WIDTH
Q_TR2	OUT	Std_Logic_Vector	WIDTH

CCwPE_TMR

The CCwPE_TMR macro is a parameterized triple redundant up counter with asynchronous set and clock enable. It is functionally equivalent to the CC16PE Virtex library macro. The generic map property Width specifies the data width of the counter's output.

Table 7: CCwPE_TMR Port Definitions

Port Name	Direction	Type	Width
CLK	IN	Std_Logic_Vector	3
PRE	IN	Std_Logic_Vector	3
CE	IN	Std_Logic_Vector	3
USERGND	IN	Std_Logic_Vector	3
TC	OUT	Std_Logic_Vector	3
CEO	OUT	Std_Logic_Vector	3
Q_TR0	OUT	Std_Logic_Vector	WIDTH
Q_TR1	OUT	Std_Logic_Vector	WIDTH
Q_TR2	OUT	Std_Logic_Vector	WIDTH

CCwRE_TMR

The CCwRE_TMR macro is a parameterized triple redundant up counter with synchronous clear and clock enable. It is functionally equivalent to the CC16RE Virtex library macro. The generic map property Width specifies the data width of the counter's output.

Table 8: CCwRE_TMR Port Definitions

Port Name	Direction	Type	Width
CLK	IN	Std_Logic_Vector	3
R	IN	Std_Logic_Vector	3
CE	IN	Std_Logic_Vector	3
USERGND	IN	Std_Logic_Vector	3
TC	OUT	Std_Logic_Vector	3
CEO	OUT	Std_Logic_Vector	3

Table 8: CCwRE_TMR Port Definitions (Continued)

Port Name	Direction	Type	Width
Q_TR0	OUT	Std_Logic_Vector	WIDTH
Q_TR1	OUT	Std_Logic_Vector	WIDTH
Q_TR2	OUT	Std_Logic_Vector	WIDTH

CCwSE_TMR

The CCwSE_TMR macro is a parameterized triple redundant up counter with synchronous set and clock enable. It is functionally equivalent to the CC16SE Virtex library macro. The generic map property Width specifies the data width of the counter's output.

Table 9: CCwSE_TMR Port Definitions

Port Name	Direction	Type	Width
CLK	IN	Std_Logic_Vector	3
S	IN	Std_Logic_Vector	3
CE	IN	Std_Logic_Vector	3
USERGND	IN	Std_Logic_Vector	3
TC	OUT	Std_Logic_Vector	3
CEO	OUT	Std_Logic_Vector	3
Q_TR0	OUT	Std_Logic_Vector	WIDTH
Q_TR1	OUT	Std_Logic_Vector	WIDTH
Q_TR2	OUT	Std_Logic_Vector	WIDTH

CCwLCPE_TMR

The CCwLCPE_TMR macro is a parameterized triple redundant UP/DOWN counter with asynchronous reset, preset, load, and synchronous clock enable. The generic map property *Width* specifies the data width of the counter's output. The input *VALUE* specifies an asynchronous initialization. If *VALUE* is connected to a signal or is left unconnected the counter will have an all zero default initialization. If *VALUE* is connected to a constant, then the default initialization of the counter will be the value of *VALUE*. When *INIT* is asserted High, all three redundant counters are asynchronously initialized to *VALUE*. An up-counter is specified by asserting *UP_DN* Low, similarly a down-counter is specified by asserting *UP_DN* High. *USERVCC* must be connected to a LOGIC 1 for normal operation (See "[V_{CC} and GND Extraction](#)" on page 21.)

Table 10: CCwLCPE_TMR Port Definitions

Port Name	Direction	Type	Width
VALUE	IN	Std_Logic_Vector	WIDTH
CLK	IN	Std_Logic_Vector	3
INIT	IN	Std_Logic_Vector	3
PRE	IN	Std_Logic_Vector	3
CLR	IN	Std_Logic_Vector	3
CE	IN	Std_Logic_Vector	3
UP_DN	IN	Std_Logic_Vector	3
USERVCC	IN	Std_Logic_Vector	3
TC	OUT	Std_Logic_Vector	3
CEO	OUT	Std_Logic_Vector	3

Table 10: CCwLCPE_TMR Port Definitions

Port Name	Direction	Type	Width
Q_TR0	OUT	Std_Logic_Vector	WIDTH
Q_TR1	OUT	Std_Logic_Vector	WIDTH
Q_TR2	OUT	Std_Logic_Vector	WIDTH

Memories

The block RAM TMR macros are similar to that shown in Figure 22. However, the block RAM macros also include address collision protection, so that the data refresh circuitry does not attempt to write to the same address currently being accessed by the user's application. Additionally, the write cycles of the data refresh for the redundant block memories are one clock cycle staggered from each other to protect against multiple data corruption.

BRAM16x256s_TMR

The BRAM16x256s_TMR macro is a triple redundant block RAM memory, single port, synchronous, 16-bit wide data bus, 256 addresses deep, 8-bit wide address bus, and built-in SEU correction. This macro is functionally equivalent to the Virtex library primitive RAMB4_S16.

Table 11: BRAM16x256s_TMR Port Definitions

Port Name	Direction	Type	Width
EN	IN	Std_Logic_Vector	3
CLK	IN	Std_Logic_Vector	3
RST	IN	Std_Logic_Vector	3
USERGND	IN	Std_Logic_Vector	3
WE	IN	Std_Logic_Vector	3
TR0_ADDR	IN	Std_Logic_Vector	8
TR1_ADDR	IN	Std_Logic_Vector	8
TR2_ADDR	IN	Std_Logic_Vector	8
TR0_DI	IN	Std_Logic_Vector	16
TR1_DI	IN	Std_Logic_Vector	16
TR2_DI	IN	Std_Logic_Vector	16
TR0_DO	OUT	Std_Logic_Vector	16
TR1_DO	OUT	Std_Logic_Vector	16
TR2_DO	OUT	Std_Logic_Vector	16

BRAM8x512s_TMR

The BRAM8x512s_TMR macro is a triple redundant block RAM memory, single port, synchronous, 8-bit wide data bus, 512 addresses deep, 9-bit wide address bus, and built-in SEU correction. This macro is functionally equivalent to the Virtex library primitive RAMB4_S8.

Table 12: BRAM8x512s_TMR Port Definitions

Port Name	Direction	Type	Width
EN	IN	Std_Logic_Vector	3
CLK	IN	Std_Logic_Vector	3
RST	IN	Std_Logic_Vector	3
USERGND	IN	Std_Logic_Vector	3
WE	IN	Std_Logic_Vector	3

Table 12: BRAM8x512s_TMR Port Definitions (Continued)

Port Name	Direction	Type	Width
TR0_ADDR	IN	Std_Logic_Vector	9
TR1_ADDR	IN	Std_Logic_Vector	9
TR2_ADDR	IN	Std_Logic_Vector	9
TR0_DI	IN	Std_Logic_Vector	8
TR1_DI	IN	Std_Logic_Vector	8
TR2_DI	IN	Std_Logic_Vector	8
TR0_DO	OUT	Std_Logic_Vector	8
TR1_DO	OUT	Std_Logic_Vector	8
TR2_DO	OUT	Std_Logic_Vector	8

BRAM4x1024s_TMR

The BRAM4x1024s_TMR macro is a triple redundant block RAM memory, single port, synchronous, 4-bit wide data bus, 1024 addresses deep, 10-bit wide address bus, and built-in SEU correction. This macro is functionally equivalent to the Virtex library primitive RAMB4_S4.

Table 13: BRAM4x1024s_TMR Port Definitions

Port Name	Direction	Type	Width
EN	IN	Std_Logic_Vector	3
CLK	IN	Std_Logic_Vector	3
RST	IN	Std_Logic_Vector	3
USERGND	IN	Std_Logic_Vector	3
WE	IN	Std_Logic_Vector	3
TR0_ADDR	IN	Std_Logic_Vector	10
TR1_ADDR	IN	Std_Logic_Vector	10
TR2_ADDR	IN	Std_Logic_Vector	10
TR0_DI	IN	Std_Logic_Vector	4
TR1_DI	IN	Std_Logic_Vector	4
TR2_DI	IN	Std_Logic_Vector	4
TR0_DO	OUT	Std_Logic_Vector	4
TR1_DO	OUT	Std_Logic_Vector	4
TR2_DO	OUT	Std_Logic_Vector	4

BRAM2x2048s_TMR

Triple redundant block RAM memory, single port synchronous 2-bit wide data bus, 2048 addresses deep, 11-bit wide address bus, and built-in SEU correction. This macro is functionally equivalent to the Virtex library primitive RAMB4_S2.

Table 14: BRAM2x2048s_TMR Port Definitions

Port Name	Direction	Type	Width
EN	IN	Std_Logic_Vector	3
CLK	IN	Std_Logic_Vector	3
RST	IN	Std_Logic_Vector	3
USERGND	IN	Std_Logic_Vector	3
WE	IN	Std_Logic_Vector	3

Table 14: BRAM2x2048s_TMR Port Definitions (Continued)

Port Name	Direction	Type	Width
TR0_ADDR	IN	Std_Logic_Vector	11
TR1_ADDR	IN	Std_Logic_Vector	11
TR2_ADDR	IN	Std_Logic_Vector	11
TR0_DI	IN	Std_Logic_Vector	2
TR1_DI	IN	Std_Logic_Vector	2
TR2_DI	IN	Std_Logic_Vector	2
TR0_DO	OUT	Std_Logic_Vector	2
TR1_DO	OUT	Std_Logic_Vector	2
TR2_DO	OUT	Std_Logic_Vector	2

BRAM1x4096s_TMR

The BRAM1x4096s_TMR macro is a triple redundant block RAM memory, single port, synchronous, one-bit wide data bus, 4096 addresses deep, 12-bit wide address bus, and built-in SEU correction. This macro is functionally equivalent to the Virtex library primitive RAMB4_S1.

Table 15: BRAM1x4096s_TMR Port Definitions

Port Name	Direction	Type	Width
EN	IN	Std_Logic_Vector	3
CLK	IN	Std_Logic_Vector	3
RST	IN	Std_Logic_Vector	3
USERGND	IN	Std_Logic_Vector	3
WE	IN	Std_Logic_Vector	3
TR0_ADDR	IN	Std_Logic_Vector	12
TR1_ADDR	IN	Std_Logic_Vector	12
TR2_ADDR	IN	Std_Logic_Vector	12
TR0_DI	IN	Std_Logic_Vector	1
TR1_DI	IN	Std_Logic_Vector	1
TR2_DI	IN	Std_Logic_Vector	1
TR0_DO	OUT	Std_Logic_Vector	1
TR1_DO	OUT	Std_Logic_Vector	1
TR2_DO	OUT	Std_Logic_Vector	1

CLKDLLs

Clk_DLL_TMR

The Clk_DLL_TMR macro is a triple redundant CLKDLL with built-in upset detection and resynchronization. The Clk_DLL_TMR macro provides all the same functional features of the Virtex library primitive CLKDLL. This macro also contains an upset detection circuit to recover DLL lock for each individual redundant DLL. When one of the three redundant clock domains are brought out of synchronization, or one of the DLLs becomes inactive, by an SEU, the DLL in the affected redundant domain is automatically reset and resynchronized. The generic map property **DIV** specifies the division factor for the CLKDV output. See the *Xilinx Libraries Guide* for acceptable DIV values and complete functional description.

Table 16: Clk_DLL_TMR Port Definitions

Port Name	Direction	Type	Width
CLKIN0	IN	Std_Logic	1
CLKIN1	IN	Std_Logic	1
CLKIN2	IN	Std_Logic	1
RST0	IN	Std_Logic	1
RST1	IN	Std_Logic	1
RST2	IN	Std_Logic	1
CLK_TR0	OUT	Std_Logic	1
CLK_TR1	OUT	Std_Logic	1
CLK_TR2	OUT	Std_Logic	1
CLK_2X0	OUT	Std_Logic	1
CLK_2X1	OUT	Std_Logic	1
CLK_2X2	OUT	Std_Logic	1
CLK_2X0	OUT	Std_Logic	1
CLK_2X1	OUT	Std_Logic	1
CLK_2X2	OUT	Std_Logic	1

Instantiating XVRWARE TMR Macros in VHDL

TMR Counter

A complete VHDL example of using an XVRWARE Counter is provided in `xvrware\synplicity\examples\Counter32_tmr.vhd`. The following is an excerpt from this example.

Component Declaration

Architecture Virtex_TMR of Counter32_tmr is

```

component xvr_ccwre_tmr
generic(width : integer);
port(CLK : in std_logic_vector(2 downto 0);
R : in std_logic_vector(2 downto 0);
CE : in std_logic_vector(2 downto 0);
USERGND : in std_logic_vector(2 downto 0);
USERVCC : in std_logic_vector(2 downto 0);
TC : out std_logic_vector(2 downto 0);
CEO : out std_logic_vector(2 downto 0);
Q_TR0 : out std_logic_vector(width-1 downto 0);
Q_TR1 : out std_logic_vector(width-1 downto 0);
Q_TR2 : out std_logic_vector(width-1 downto 0));
end component;

```

Since the redundant clock signals of the counter are vectored, if the clock signals of the parent component are individual, then a vector must be created for port mapping.

```

Signal Count32_tr0 : std_logic_vector(31 downto 0);
Signal Count32_tr1 : std_logic_vector(31 downto 0);
Signal Count32_tr2 : std_logic_vector(31 downto 0);

```

```
Begin
```

Instantiation

The **width** generic parameter must be set to any integer value for instantiation. The following instance specifies a 32-bit counter. Subsequently, the triple redundant output ports must be mapped to 32-bit vector signals.

```

counter: xvr_ccwre_tmr
generic map(width=>32)
port map(CLK=>CLK, R=>RST, CE=>CE, USERGND=>RST, USERVCC=>CE,
Q_TR0=>Count32_tr0, Q_TR1=>Count32_tr1, Q_TR2=>Count32_tr2);

```

This instantiation makes use of the parent component's RST and CE signals to tie off the USERVCC and USERGND ports.

If the count values are also the outputs to the FPGA design, then their final voting and recombination are accomplished with the OUTPUT_TMR macro. One macro must be instantiated for each bit of the output bus. This is most easily accomplished with a *Generate* statement.

```

L0: For I in 0 to 31 generate
outputs: output_tmr
port map(TR0=>Count32_tr0(I), TR1=>Count32_tr1(I),
TR2=>Count32_tr2(I), V0=>Count_tr0(I), V1=>Count_tr1(I), V2=>Count_tr2(I));
end generate;

End Virtex_TMR;

```

TMR Block RAM

The TMR block RAM macros provide a synchronous single-port memory block function. The following example is taken from `xvrrware\synplicity\src\block RAM_TMR.vhd`. The component declaration for the BRAM8x512s_TMR macro is as follows:

Component Declaration

```

ARCHITECTURE MITIGATED OF block RAM_TMR IS

component bram8x512s_tmr
generic (INIT_00, INIT_01, INIT_02, INIT_03,
INIT_04, INIT_05, INIT_06, INIT_07,
INIT_08, INIT_09, INIT_0A, INIT_0B,
INIT_0C, INIT_0D, INIT_0E, INIT_0F : string :=
"00000000000000000000000000000000000000000000000000000000000000000000");
port ( EN : in std_logic_vector(2 downto 0);
CLK : in std_logic_vector(2 downto 0);
RST : in std_logic_vector(2 downto 0);
WE : in std_logic_vector(2 downto 0);
USERGND : in std_logic_vector(2 downto 0);
tr0_ADDR : in std_logic_vector(8 downto 0);
tr1_ADDR : in std_logic_vector(8 downto 0);
tr2_ADDR : in std_logic_vector(8 downto 0);
tr0_DI : in std_logic_vector(7 downto 0);
tr1_DI : in std_logic_vector(7 downto 0);
tr2_DI : in std_logic_vector(7 downto 0);
tr0_DO : out std_logic_vector(7 downto 0);
tr1_DO : out std_logic_vector(7 downto 0);
tr2_DO : out std_logic_vector(7 downto 0));
end component;

```

The INIT generic attributes are used to specify initial data values for the memory content. See either the *Xilinx Library Guide* or the *Synplify Help* for data formats. If the INIT values are left unspecified, all values will default to "0."

Instantiation

The following instantiation initializes the data content to a binary count sequence. The INIT values are propagated to all three redundant memory elements.

```

BRV: bram8x512s_tmr
generic map (
INIT_00 => "201F1E1D1C1B1A191817161514131211
100F0E0D0C0B0A090807060504030201",

```



```

INIT_01 => "403F3E3D3C3B3A393837363534333231
302F2E2D2C2B2A292827262524232221",
INIT_02 => "605F5E5D5C5B5A595857565554535251
504F4E4D4C4B4A494847464544434241",
INIT_03 => "807F7E7D7C7B7A797877767574737271
706F6E6D6C6B6A696867666564636261",
INIT_04 => "A09F9E9D9C9B9A999897969594939291
908F8E8D8C8B8A898887868584838281",
INIT_05 => "C0BFBEBCBBBAB9B8B7B6B5B4B3B2B1
B0AFAEADACABAAA9A8A7A6A5A4A3A2A1",
INIT_06 => "E0DFDEDDDCDBDAD9D8D7D6D5D4D3D2D1
D0CFCECDCCBCAC9C8C7C6C5C4C3C2C1",
INIT_07 => "00FFFEFDFCFBFAF9F8F7F6F5F4F3F2F1
F0EFEEDCEBEAE9E8E7E6E5E4E3E2E1",
INIT_08 => "201F1E1D1C1B1A191817161514131211
100F0E0D0C0B0A090807060504030201",
INIT_09 => "403F3E3D3C3B3A393837363534333231
302F2E2D2C2B2A292827262524232221",
INIT_0A => "605F5E5D5C5B5A595857565554535251
504F4E4D4C4B4A494847464544434241",
INIT_0B => "807F7E7D7C7B7A797877767574737271
706F6E6D6C6B6A696867666564636261",
INIT_0C => "A09F9E9D9C9B9A999897969594939291
908F8E8D8C8B8A898887868584838281",
INIT_0D => "C0BFBEBCBBBAB9B8B7B6B5B4B3B2B1
B0AFAEADACABAAA9A8A7A6A5A4A3A2A1",
INIT_0E => "E0DFDEDDDCDBDAD9D8D7D6D5D4D3D2D1
D0CFCECDCCBCAC9C8C7C6C5C4C3C2C1",
INIT_0F => "00FFFEFDFCFBFAF9F8F7F6F5F4F3F2F1
F0EFEEDCEBEAE9E8E7E6E5E4E3E2E1")
port map (CLK=>CLK, RST=>RST, USERGND=>USERGND, EN=>EN, WE=>WE,
tr0_ADDR=>ADDR_TR0, tr0_DI=>DATA_IN_TR0, tr0_DO=>DATA_OUT_TR0,
tr1_ADDR=>ADDR_TR1, tr1_DI=>DATA_IN_TR1, tr1_DO=>DATA_OUT_TR1,
tr2_ADDR=>ADDR_TR2, tr2_DI=>DATA_IN_TR2, tr2_DO=>DATA_OUT_TR2);

```

The block RAM TMR macros require a USERGND connection. This connection **should not** be made to the block RAM's RST signal as this can disrupt the data refresh circuitry. However, connection to an active High global FPGA reset signal is adequate, if a dedicated USERGND signal is not available. See [V_{CC} and GND Extraction, page 21](#) for more details.

TMR CLKDLL

The following example is taken from
xvrrware\synplicity\examples\C1k_Mgr_TMR.vhd.

Component Declaration

```

component Clk_DLL_TMR
generic (DIV : string);
port (
CLKIN0 : in std_logic;
CLKIN1 : in std_logic;
CLKIN2 : in std_logic;
RST0 : in std_logic;
RST1 : in std_logic;
RST2 : in std_logic;
CLKDV0 : out std_logic;
CLKDV1 : out std_logic;
CLKDV2 : out std_logic;
CLK2X0 : out std_logic;
CLK2X1 : out std_logic;
CLK2X2 : out std_logic;
CLK_TR0 : out std_logic;

```

```

CLK_TR1 : out std_logic;
CLK_TR2 : out std_logic);
end component;

```

Although a CLKDLL requires an IBUFG on its CLKIN port and a BUFG to feedback the clock output from CLK0 to CLKFB, the Clk_DLL_TMR macro already provides these, so these instantiations are not necessary.

The CLK_TR n output ports drive a BUFG and thus can be directly connected to any clock loads in the FPGA design. In order to use one of the other output selections, a BUFGS should be instantiated in series.

The CLKDV output provides a clock division of the reference input clock specified by the DIV attribute. See the *Xilinx Libraries Guide* for value options.

Instantiation

```

TDLL: Clk_DLL_TMR
generic map (DIV=>"2")
port map (CLKIN0=>CLKIN0, CLKIN1=>CLKIN1, CLKIN2=>CLKIN2, RST0=>RST0,
RST1=>RST1, RST2=>RST2, CLK_TR0=>CLK_TR0, CLK_TR1=>CLK_TR1,
CLK_TR2=>CLK_TR2, CLKDV0=>CLKDV0, CLKDV1=>CLKDV1, CLKDV2=>CLKDV2,
CLK2X0=>CLK2X0, CLK2X1=>CLK2X1, CLK2X2=>CLK2X2);

```

The instantiation above specifies a clock division of two. Therefore, the CLKDV n outputs will be one half the CLKIN n frequency, but phase-locked with CLK_TR n .

Other Examples in VHDL

TMR State-Machines

Finite State-Machines (FSM) are an inevitable aspect of digital design. FSM circuitry implements sequential processing, sequencing control, and decision-making algorithms. Development methods for FSM circuits range from explicit *Case* statements to FSM compilers.

Like counters, state-machines create registered feedback logic loops that must be replicated and voted in order to assure reliability against SEUs. The TMR implementation of such circuits might seem a bit tedious, but its effectiveness is well worth the effort.

The following example is a simple four-state state-machine with two “event” inputs: A and B. The bubble diagram is shown in **Figure 30**.

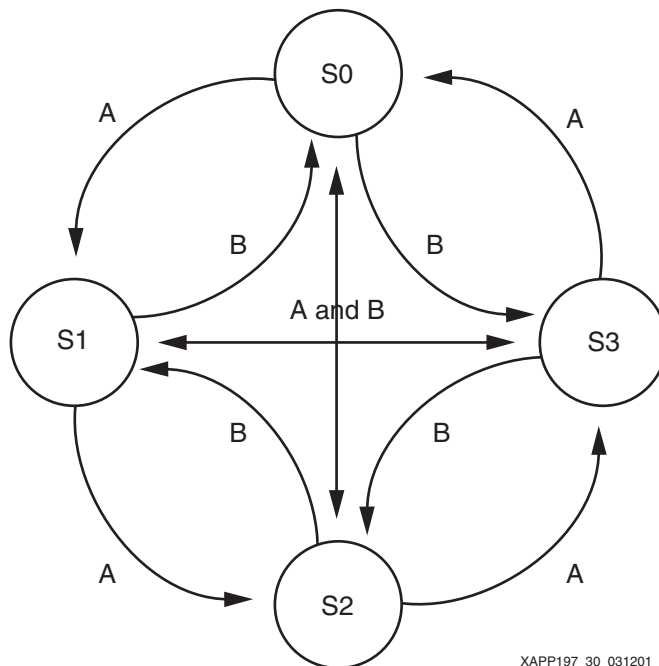


Figure 30: Dual Event State Machine Bubble Diagram

When Event A is active the state machine proceeds to the next state number. When Event B is active the state machine proceeds to the previous state number. When both Event A and Event B are active the state machine transitions to the opposite state. When neither event are active the state machine stays in its current state.

A VHDL representation of this circuit is shown in

xvrvare\synplicity\examples\Dual_Event_FSM.vhd.

A One-Hot encoding scheme has been arbitrarily selected for the FSM. Symbolic encoding will not work for implementing TMR state-machines, because the majority voters must be inserted into each register feedback path. Therefore, state-machine encoding must be explicit. The registered State signal forms a registered logic loop. Therefore, this is the insertion point for majority voters when replicating the triple redundancy. In the design, the FSM states are decoded into separate outputs that control external switches. These are throughput logic structures and as such do not require majority voters. However, since they are FPGA outputs, OUTPUT_TMR macros will be inserted along their FPGA exit path. The TMR version of this design is shown in **xvrvare\synplicity\examples\Dual_Event_FSM_TMR.vhd** and is as follows:

Notes:

1. There are further text comments between code segments.)

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_logic_unsigned.all;

Entity dual_event_fsm_tmr Is
port (Switch_Control_TR0 : out std_logic_vector(3 downto 0);
      Switch_Control_TR1 : out std_logic_vector(3 downto 0);
      Switch_Control_TR2 : out std_logic_vector(3 downto 0);
      Event_A, Event_B : in std_logic_vector(2 downto 0);
      Clk, Rst, Ce : in std_logic_vector(2 downto 0);
      user_gnd : in std_logic_vector(2 downto 0)); -- Routed GND signal.
End dual_event_fsm_tmr;
```

Comments: There are now three redundant versions of every input and output. Individual bit signals have become three bit vectors and logic vectors have been replicated.

```
Architecture rtl Of dual_event_fsm_tmr Is

component output_tmr
port (
  TR0 : in std_logic;
  TR1 : in std_logic;
  TR2 : in std_logic;
  V0 : out std_logic;
  V1 : out std_logic;
  V2 : out std_logic);
end component;
```

Comments: The BUFT style voter has been arbitrarily selected for this implementation.

```
component TRV_BUFT
port (
  TR0 : in std_logic;
  TR1 : in std_logic;
  TR2 : in std_logic;
  V : out std_logic);
end component;
```

Comments: The original signal "State" has been expanded to a three vector array to accommodate the triple redundancy. The Array has two variable indicies. The first index specifies the redundant domain and the second specifies the bit position of the vector.

```
-- Use Arrays to create redundant vectors.
subtype Vector2 is Std_Logic_Vector(1 downto 0);
```

```

Type Vector2Array is ARRAY (0 to 2) of Vector2;

subtype Vector4 is Std_Logic_Vector(3 downto 0);
Type Vector4Array is ARRAY (0 to 2) of Vector4;

Signal State, StateV : Vector2Array;
Signal Sw_Ctl : Vector4Array;

```

Comments: The output_tmr macro is inserted on the output paths.

```

-- Instantiate Triple Redundant Output Module.
L0: For I in 0 to 3 Generate
  SW0: output_tmr
    port map(TR0=>Sw_Ctl(0)(I), TR1=>Sw_Ctl(1)(I), TR2=>Sw_Ctl(2)(I),
      V0=>Switch_Control_TR0(I), V1=>Switch_Control_TR1(I),
      V2=>Switch_Control_TR2(I));
end generate;

```

Comments: The TRV_BUFT voters are inserted into the registered feedback paths. One voter is inserted for each bit in each redundant path. Since this is a **two-bit** state machine with **triple** redundancy, **six** voters are instantiated. Since each voter uses three BUFTs, the total implementation will use 18 BUFTs. Two Generate statements are used to insert the voters. The first generate loop (K) specifies the redundant domain, while the second (I) specifies the bit position.

```

-- Instantiate Voters into feedback paths.
L1: For K in 0 to 2 Generate -- K is the redundant domain.
  L2: For I in 0 to 1 Generate -- I is the bit position.
    SV: TRV_BUFT -- Or use TRV_LUT for
      port map(TR0=>State(0)(I), TR1=>State(1)(I), TR2=>State(2)(I),
        V=>StateV(K)(I));
    end generate;
  end generate;
end generate;

```

Comments: The redundancies for the decoders are implemented by embedding the original code inside a LOOP statement (just as Generate was used for redundant instances) and specifying the redundant domain (bit index for signals and vector index for arrays) with a variable (K). The CASE sensitivity was previously the State signal in the original code, but is now the triple redundant arrayed majority vote signal StateV (output of the majority voters).

```

-- CASE sensitivity for Voted signal.
SW_DEC: process (StateV)
begin
For K in 0 to 2 Loop -- Loop to generate redundancy.
  CASE StateV(K) Is
    When "00" =>
      Sw_Ctl(K) <= "0001";
    When "01" =>
      Sw_Ctl(K) <= "0010";
    When "11" =>
      Sw_Ctl(K) <= "0100";
    When "10" =>
      Sw_Ctl(K) <= "1000";
    When others => null;
  End Case;
End Loop;
end process;

```

Comments: Similarly, the redundancies for the state machine are implemented with a LOOP statement. Notice carefully the insertion of an additional asynchronous initialization condition. The original code specified a synchronous reset function, but not an asynchronous condition. This is not a problem if this function of the primitive Virtex flip-flop is utilized; however, some

synthesis tools may not make use of this feature leaving the initialization control of the flip-flops unspecified. (See "[V_{CC} and GND Extraction](#)" on page 21.) The USER_GND signal is, therefore, used to make this specification.

```

FSM: process(Clk)
begin
For K in 0 to 2 Loop -- Loop to generate redundancy.
If (user_gnd(K) = '1') Then -- Tie off FF init pin.
  State(K) <= "00";
ElsIf (Clk(K)'event and Clk(K)='1') Then
  If (Rst(K)='1') Then
    State(K) <= "00";
  Elsif (Ce(K)='1') Then
    CASE StateV(K) Is
    When "00" =>
      If (Event_A(K) = '1' and Event_B(K) = '1') Then
        State(K) <= "11";
      Elsif ...

    End If;
  When others => null;
  End Case;
  End If;
End If;
End Loop;
end process;

```

Comments: Although other mitigation methods can warn against the use of a 'null' expression for the construction of Finite State-Machines, it is not a point of concern for this method. The redundant feedback paths will correct any redundant register that is upset to an illegal state.

Conclusion

Implementing proper Triple Module Redundancy in an FPGA design will harden that design from any single configuration bit upset. Combining this technique with the partial reconfiguration methods described in XAPP216 provides an SEU Mitigation method that obtains an absolute zero SEU error rate. However, this error rate does not take into account any device-level, single-event functional interrupts. See data sheet for SEFI cross-sections.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
11/01/01	1.0	Initial Xilinx release (with corrected FTP site information).
07/06/06	1.0.1	Updated link to reference design files.